

# Docker Automation with Dockerfiles (Windows)

Tuesday, August 29, 2017 12:25 AM

---

Learn how to automate the build of a custom Windows-based Docker image from a Dockerfile.

This workshop will show you how to automate the building and configuring of a Windows-based Docker image by utilizing Dockerfiles. You will construct a Dockerfile by mimicking a production environment configuration. You'll also learn some of the options for a Dockerfile configuration.

### **What You Will Learn**

- Constructing a Dockerfile for Windows-based Builds
- Various Dockerfile Configuration Options
- Building a Docker Image from a Dockerfile

### **Ideal Audience**

- IT Managers
- Developers and Software Architects
- Configuration and Change Managers
- DevOps Engineers

# Overview

This workshop will show you how to automate the building and configuring of a Windows-based Docker image by utilizing Dockerfiles. You will construct a Dockerfile by mimicking a production environment configuration. You'll also learn some of the options for a Dockerfile configuration.

**Time Estimate:** 2 hours

# Requirements

## Setup Requirements

The following workshop will require that you use a Remote Desktop client in order to connect to a remote machine. If you are using a Mac, then [download](#) the Microsoft Remote Desktop client.

## Additional Requirements

For the following workshop, you will need a subscription (trial or paid) to Microsoft Azure. Please see the [next](#) page for how to create a trial subscription, if necessary.

# Azure Registration

## Azure

We need an active Azure subscription in order to perform this workshop. There are a few ways to accomplish this. If you already have an active Azure subscription, you can skip the remainder of this page. Otherwise, you'll either need to use an Azure Pass or create a trial account. The instructions for both are below.

## Azure Pass

If you've been provided with a voucher, formally known as an Azure Pass, then you can use that to create a subscription. In order to use the Azure Pass, direct your browser to <https://www.microsoftazurepass.com> and, following the prompts, use the code provided to create your subscription.

## Trial Subscription

Direct your browser to <https://azure.microsoft.com/en-us/free/> and begin by clicking on the green button that reads **Start free**.

1. In the first section, complete the form in its entirety. Make sure you use your *real* email address for the important notifications.
2. In the second section, enter a *real* mobile phone number to receive a text verification number. Click send message and re-type the received code.
3. Enter a valid credit card number. **NOTE:** You will *not* be charged. This is for verification of identity only in order to comply with federal regulations. Your account statement may see a temporary hold of \$1.00 from Microsoft, but, again, this is for verification only and will "fall off" your account within 2-3 banking days.
4. Agree to Microsoft's Terms and Conditions and click **Sign Up**.

This may take a minute or two, but you should see a welcome screen informing you that your subscription is ready. Like the Office 365 trial above, the Azure subscription is good for up to \$200 of resources for 30 days. After 30 days, your subscription (and resources) will be suspended unless you convert your trial subscription to a paid one. And, should you choose to do so, you can elect to use a different credit card than the one you just entered.

Congratulations! You've now created an Office 365 tenant; an Azure tenant and subscription; and, have linked the two together.

# Introduction

## Objective

In this workshop, you will first build a 'production' web server environment on the actual virtual machine. You will then take those same steps and replicate them in a Dockerfile for building a containerized version of your VM.

The steps in this workshop are not extremely tedious. However, they are broken out into individual pages for a couple of reasons. First, it is to simplify the process and aid you in your comprehension. Second, it is for the purpose of you seeing the actual steps of building the production virtual machine so that you comprehend what you are doing as you add each step to the Dockerfile.




# Create Virtual Machine

## Objective

All of our work in this workshop, with the exception of the small Azure configuration at the end, will be performed on a single virtual machine. Let's get started creating that VM.

## Create a Resource Group

In order to create resources, we need a *Resource Group* to place them in.


1. If you are not there already, go ahead and click on the **Resource Groups**  in the Azure Portal to open the Resource Groups blade.
2. At the top of the Resource Groups blade, click on **Add** . This will open a panel that asks for some basic configuration settings.
3. Complete the configuration settings with the following:
  - Resource group name: **azworkshops\_dockerfile\_win\_demo**
  - Subscription: **<choose your subscription>**
  - Resource group location: **<choose your location>**
4. *<Optional>* Check *Pin to dashboard* at the bottom of the panel.
5. Click **Create**.
6. It should only take a second for the resource group to be created. Once you click create, the configuration panel closes and returns you to the list of available resource groups. Your recently created group may not be visible in the list. Clicking on **Refresh**  at the top of the Resource Groups blade should display your new resource group.

**NOTE:** When you create a resource group, you are prompted to choose a location. Additionally, as you create individual resources, you will also be prompted to choose locations. The location of resource groups and their resources can be different. This is because resource groups store *metadata* describing their contained resources; and, due to some types of compliance that your company may adhere to, you may need to store that metadata in a different location than the resources themselves. For example, if you are a US-based company, you may choose to keep the metadata state-side while creating resources in foreign regions to reduce latency for the end-user.















# Create a Virtual Machine

Now that we have an available resource group, let's create the actual Windows server.

1. If you are not there already, go ahead and navigate to the **azworkshops\_dockerfile\_win\_demo** resource group.
2. At the top of the blade for our group, click on **Add** . This will display the blade for the *Azure Marketplace* allowing you to deploy a number of different solutions.
3. We are interested in deploying a Windows Server 2016 Datacenter server. Therefore, in the *Search Everything* box, type in **Windows Server 2016**. This will display a couple of different versions. Choose **Windows Server 2016 Datacenter**.



4. There will be a number of solutions available, including one with containers already enabled. For the practice, we'll enable containers manually. Therefore, choose the image as highlighted in the image below.

Windows Server 2016 Datacenter			
Results			
NAME	PUBLISHER	CATEGORY	
 Windows Server 2016 Datacenter	Microsoft	Compute	
 [HUB] Windows Server 2016 Datacenter	Microsoft	Compute	
 Windows Server 2016 Datacenter	Microsoft		
 Windows Server 2016 Datacenter - with Containers	Microsoft	Compute	
 [smalldisk] Windows Server 2016 Datacenter	Microsoft	Compute	
 Windows Server 2016 Datacenter - Server Core	Microsoft	Compute	
 [smalldisk] Windows Server 2016 Datacenter - Server Core	Microsoft	Compute	
 SharePoint Server 2016 Trial	Microsoft	Compute	
 HPC Pack 2016 Head Node on Windows Server 2016	Microsoft	Compute	
 HPC Pack 2016 Compute Node on Windows Server 2016	Microsoft	Compute	
 HPC Pack 2016 Head Node on Windows Server 2012 R2	Microsoft	Compute	
 HPC Pack 2016 Compute Node on Windows Server 2012 R2	Microsoft	Compute	

5. This will display a blade providing more information about the server we have chosen. To continue creating the server, choose **Create**.

6. We are now prompted with some configuration options. There are 3 sections we need to complete and the last section is a summary of our chosen options.

#### 1. Basics

- Name: **docker-win**
- VM disk type: **SSD**
- Username: **localadmin**
- Password: **Pass@word1234**
- Confirm password: *<same as above>*
- Subscription: *<choose your subscription>*
- Resource group: **Use existing - azworkshops\_dockerfile\_win\_demo**
- Location: *<choose a location>*
- Already have a Windows Server license? **No**

#### 2. Size

- **DS1\_V2**

#### 3. Settings

- Use managed disks: **No**

- Storage account: (click on it & **Create New**)
  - Name: **dfwindata**<random number> (ex. *dfwindata123456*)  
(NOTE: This name must be *globally* unique, so it cannot already be used.)
  - Performance: **Premium**
  - Replication: **Locally-redundant storage (LRS)**
- Virtual network: <accept default> (e.g. *(new) azworkshops\_dockerfile\_win\_demo-vnet*)
- Subnet: <accept default> (e.g. *default (172.16.1.0/24)*)
- Public IP address: <accept default> (e.g. *(new) docker-win-ip*)
- Network security group (firewall): <accept default> (e.g. *(new) docker-win-nsg*)
- Extensions: **No extensions**
- Availability set: **None**
- Boot diagnostics: **Enabled**
- Guest OS diagnostics: **Disabled**
- Diagnostics storage account: (click on it & **Create New**)
  - Name: **dfwindiags**<random number> (ex. *dfwindiags123456*)
  - Performance: **Standard**
  - Replication: **Locally-redundant storage (LRS)**

4. Summary (just click **OK** to continue)

Once scheduled, it may take a minute or two for the machine to be created by Azure. Once it has been created, Azure *should* open the machine's status blade automatically.

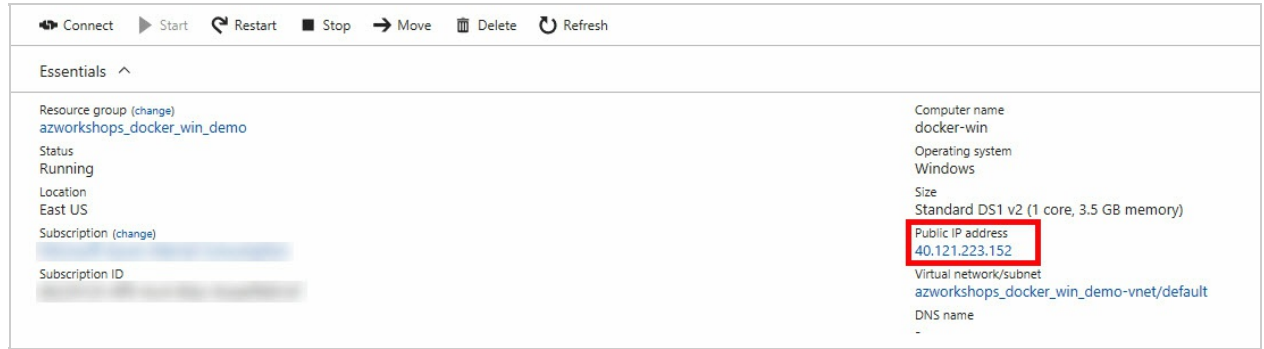
## Connect to the Virtual Machine

Once your machine has been created, we can remotely connect to it via a remote desktop protocol (RDP) client.

### Get Public IP

1. If it is not already open, navigate to the **Overview** blade of your newly created virtual machine.



2. In the top section of the blade, in the right column, you should see a **Public IP address** listed.



3. Copy the IP address.

### Connect to the Machine via Remote Desktop

To connect to the machine remotely, we need to download the Remote Desktop Protocol (RDP) profile.

1. Click on the **Overview**  to return to the general information for the **docker-win** virtual machine.
2. In the **Actions** section, click on **Connect** . This will download the RDP profile to your machine.
3. Open the profile and accept any warnings.
4. For the username, enter **\localadmin** (with the backslash). And, for the password, enter **Pass@word1234**. Click **OK**.
5. Again, accept any warnings.

Congratulations. You have successfully created and connected to your remote Windows Server 2016 server in Azure. You are now ready to install the Docker runtime.

# Install Docker

## Overview

We have just created our Windows Server 2016 server. We now need to apply any available system updates along with installing and configuring Docker to begin working with containers.

## Install Updates

Just like any other operating system, updates are periodically released to support new features and patch any potential security threats. We will apply the updates first.

1. If you have not already, connect to your remote Windows Server 2016 server and login.
2. Open a command prompt as an Administrator, type the following at the command prompt:

```
sconfig
```

3. This will open a screen like the following:

```
Administrator: Command Prompt - sconfig
Microsoft (R) Windows Script Host Version 5.812
Copyright (C) Microsoft Corporation. All rights reserved.

Inspecting system...

=====
                        Server Configuration
=====

1) Domain/Workgroup:           Workgroup:  WORKGROUP
2) Computer Name:             DOCKER-WIN
3) Add Local Administrator
4) Configure Remote Management  Enabled
5) Windows Update Settings:    DownloadOnly
6) Download and Install Updates
7) Remote Desktop:            Enabled (more secure clients only)
8) Network Settings
9) Date and Time
10) Telemetry settings         Enhanced
11) Windows Activation
12) Log Off User
13) Restart Server
14) Shut Down Server
15) Exit to Command Line

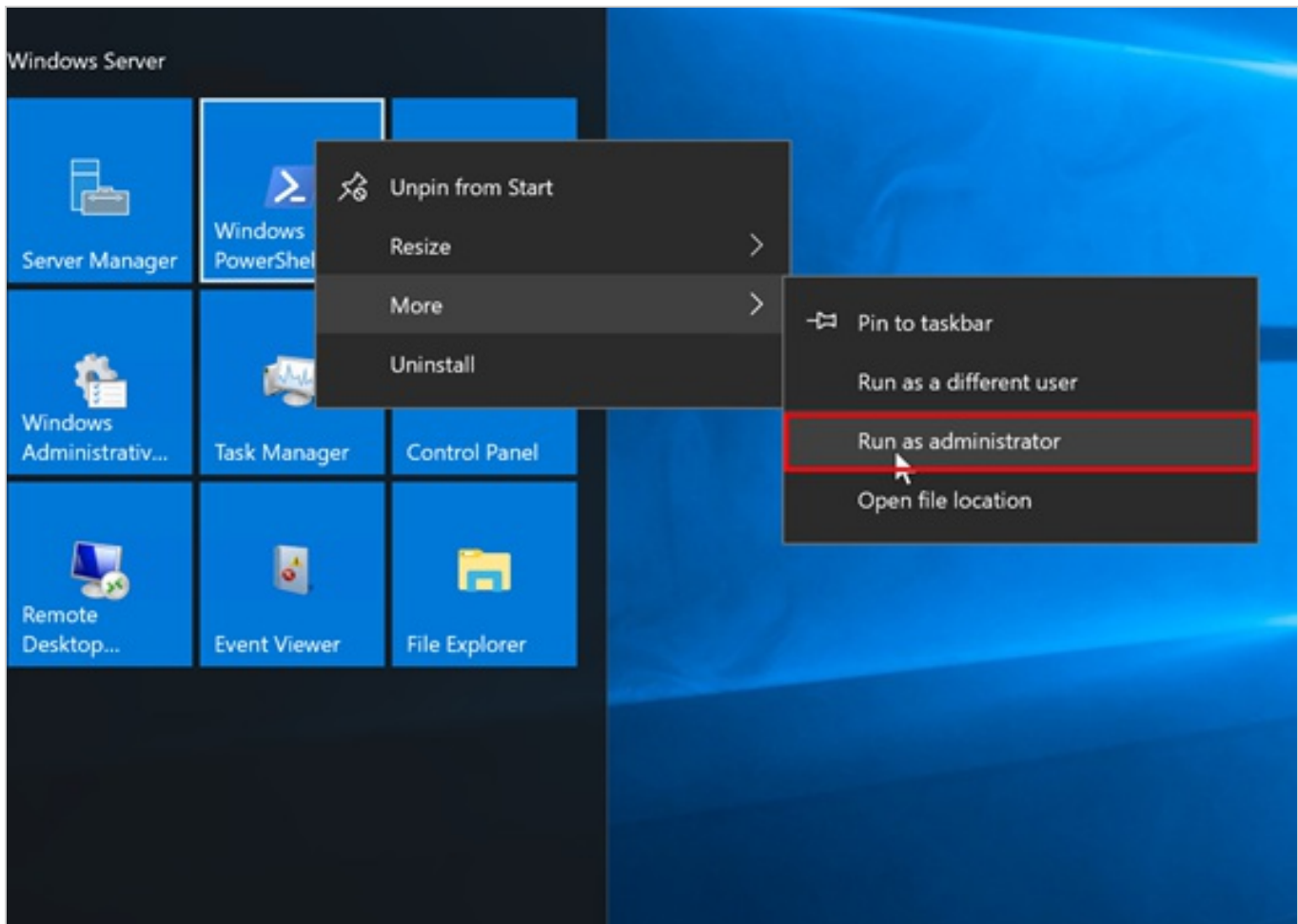
Enter number to select an option: █
```

4. Choose option 6 , then A (twice) to download and install all updates.
5. Depending on the number and size of available updates, this process may take a few minutes and could require a reboot. Now would be a good time to take a break.

## Install Docker

We now have an updated Windows operating system. We are ready to install Docker.

1. Open a PowerShell prompt as an Administrator:



2. At the prompt, type the following:

```
Install-PackageProvider -Name NuGet -MinimumVersion 2.8.5.201 -Force
Install-Module -Name DockerMsftProvider -Force
Install-Package -Name docker -ProviderName DockerMsftProvider -Force
Restart-Computer -Force
```

3. This will download the Docker engine and install it as a background service.
4. After you run the above commands, your virtual machine will reboot forcing a disconnect. Go ahead and reconnect.

### Ensure Docker Engine is Running

1. Open a PowerShell prompt as an Administrator and type the following:

```
docker version
```

2. You should see something similar to the following:

```
Client:
  Version:      17.03.1-ee-3
  API version:  1.27
  Go version:   go1.7.5
  Git commit:   3fcee33
  Built:        Thu Mar 30 19:31:22 2017
  OS/Arch:     windows/amd64

Server:
  Version:      17.03.1-ee-3
  API version:  1.27 (minimum version 1.24)
  Go version:   go1.7.5
  Git commit:   3fcee33
  Built:        Thu Mar 30 19:31:22 2017
  OS/Arch:     windows/amd64
  Experimental: false
```

3. Because the service is running, we can now use the `docker` command later in this workshop.

You've successfully installed the Docker engine.



# Install IIS

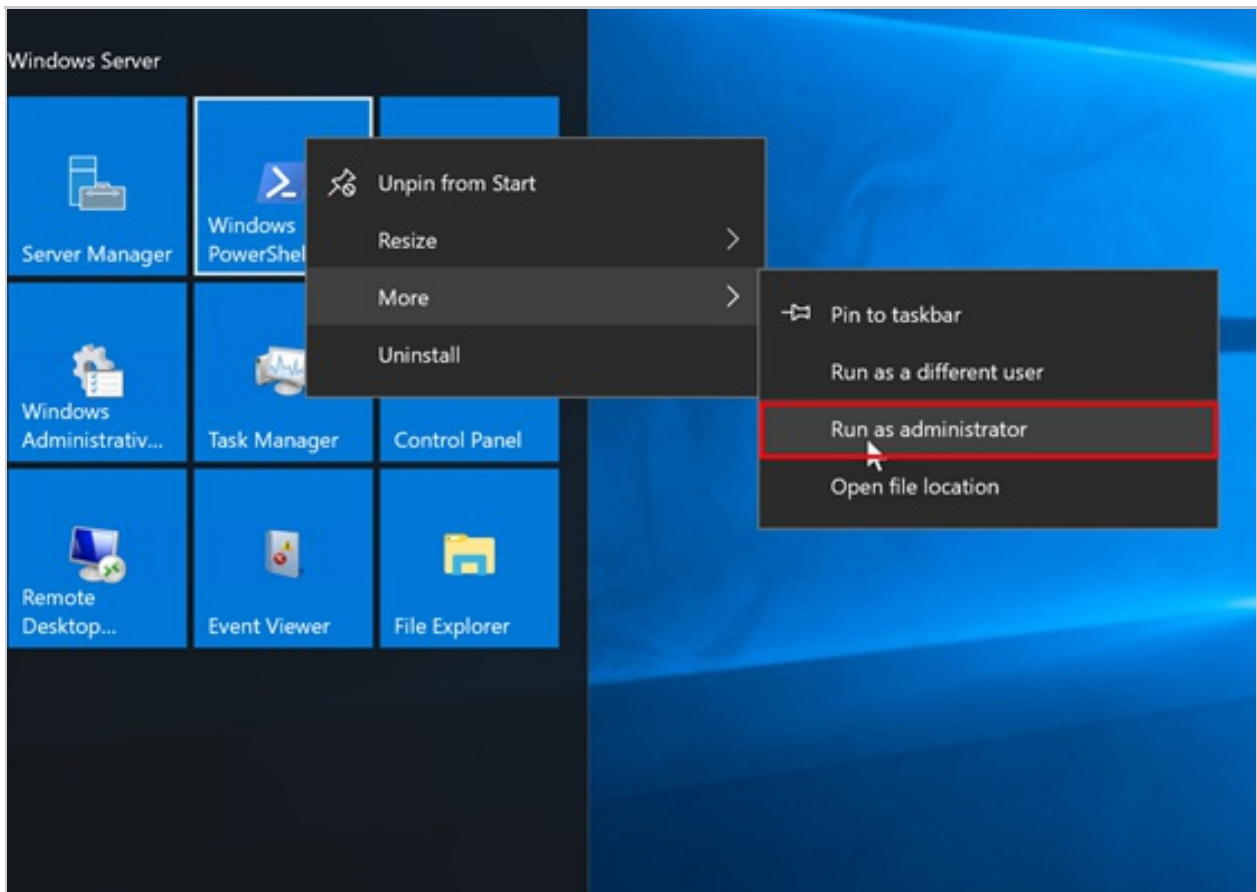
## Overview

The final two elements of preparing our Windows Server virtual machine is to install Internet Information Server (IIS) and configure the necessary port (80) in the firewall to allow HTTP requests.

## Install IIS

Since, for this example, we will be deploying and hosting a basic, static website, the standard IIS components are sufficient. We could install them through the Server Manager, but we are going to use PowerShell so that we become familiar with executing tasks for later when we need to automate this process in Docker.

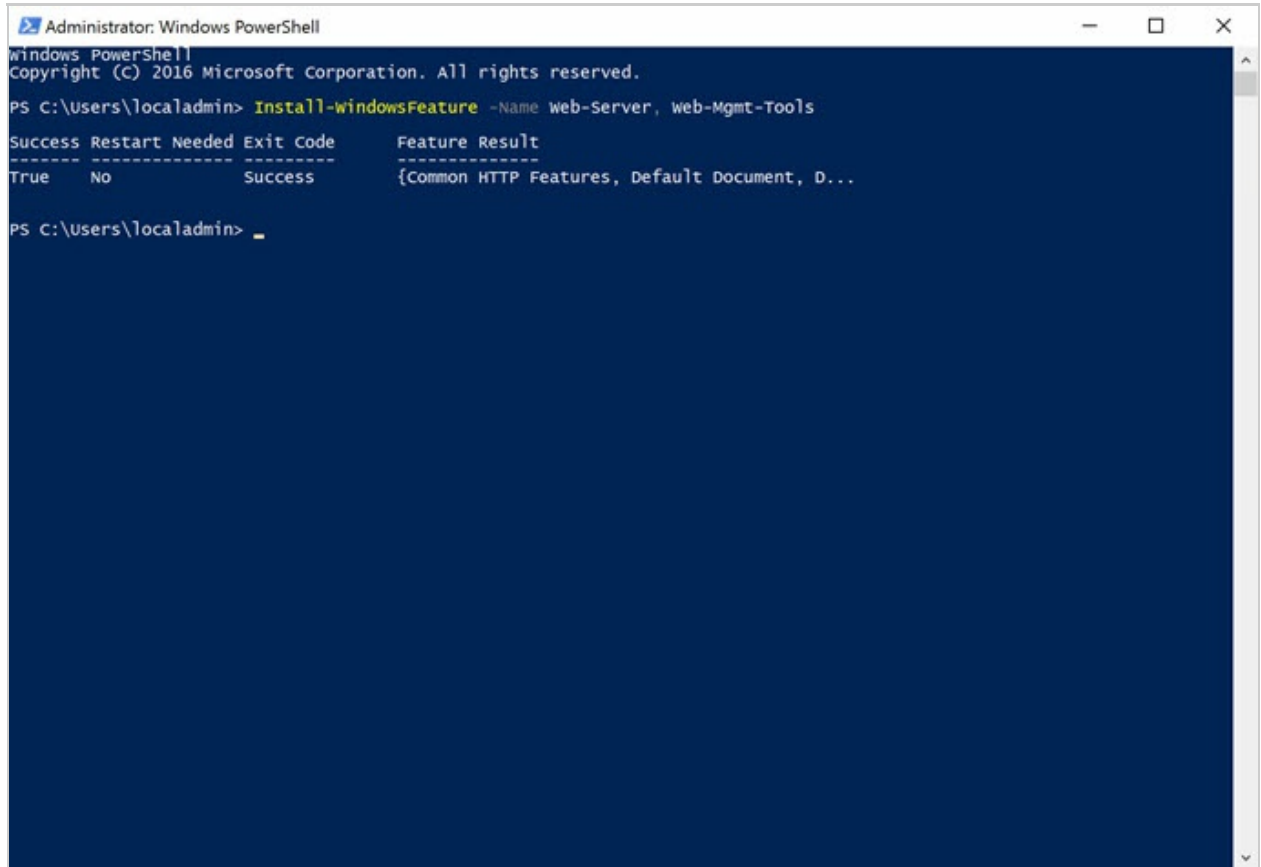
1. Open PowerShell in elevated mode (with Administrator privileges):



2. Type and execute the command:

```
Install-WindowsFeature -Name Web-Server, Web-Mgmt-Tools, NET-Framework-45-ASPNET, Web-App-Dev, Web-Net-Ext45, Web-AppInit, Web-Asp-Net45, Web-ISAPI-Ext, Web-ISAPI-Filter
```

3. You should then see the components download and install.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The terminal output is as follows:

```
windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Users\localadmin> Install-WindowsFeature -Name web-Server, web-Mgmt-Tools

Success Restart Needed Exit Code      Feature Result
-----
True      No              Success          {Common HTTP Features, Default Document, D...
```

The prompt returns to "PS C:\Users\localadmin> \_".

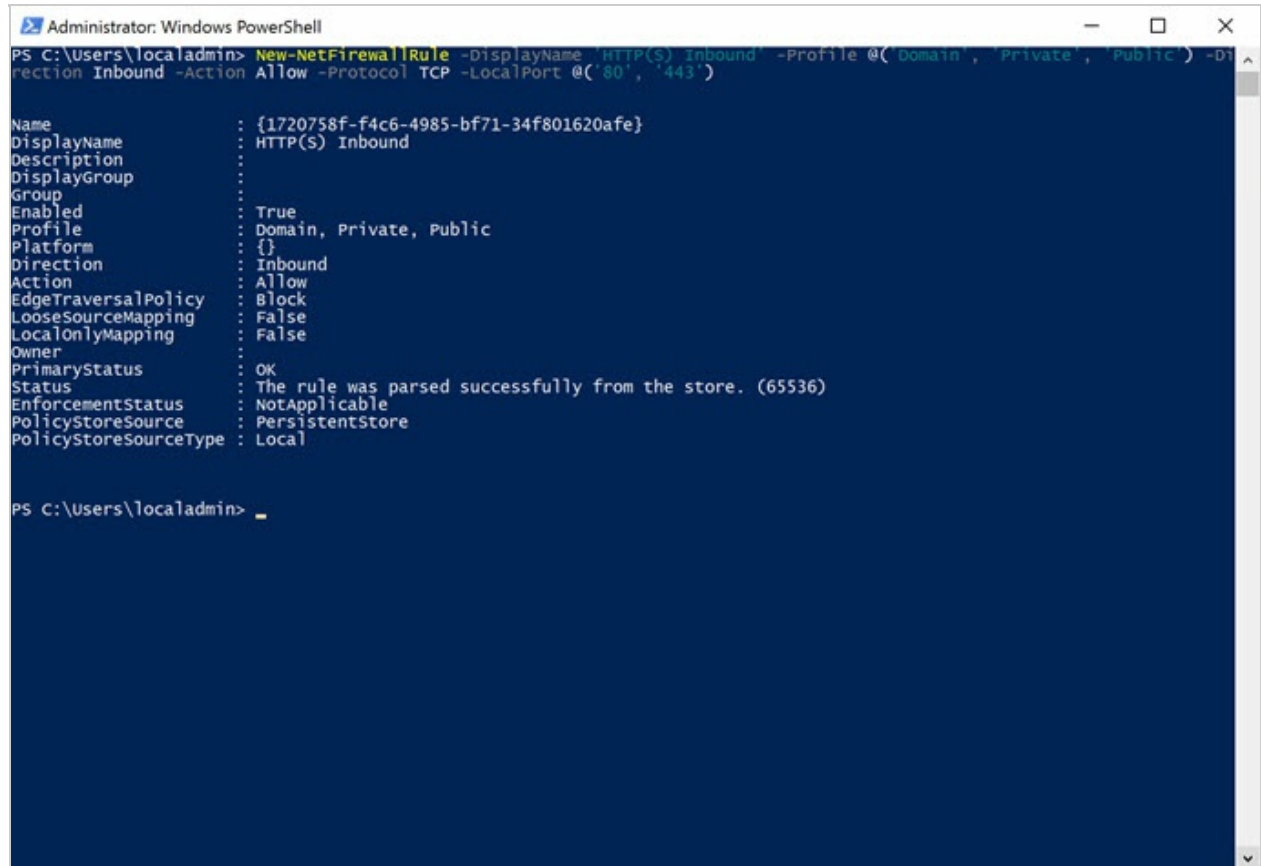
4. It shouldn't be necessary, but just to be safe, let's reset IIS to pickup the installation of any additional modules. Type the following and press Enter: `iisreset /restart`
5. We should then see some messages telling us that IIS restarted successfully.

## Configure Firewall

The last step to configuring our server is to allow IIS to serve webpages through port 80. By default, the port is blocked and so, even if IIS was running, we would not be able to access the site outside of the server, itself. We, again, are going to use PowerShell to configure the firewall.

1. If it's not already open, again open PowerShell in elevated mode.
2. Type the following command

```
New-NetFirewallRule -DisplayName 'HTTP(S) Inbound' -Profile @('Domain', 'Private', 'Public') -Direction Inbound -Action Allow -Protocol TCP -LocalPort @('80', '443')
```



```
Administrator: Windows PowerShell
PS C:\Users\localadmin> New-NetFirewallRule -DisplayName 'HTTP(S) Inbound' -Profile @('Domain', 'Private', 'Public') -Direction Inbound -Action Allow -Protocol TCP -LocalPort @('80', '443')

Name                : {1720758f-f4c6-4985-bf71-34f801620afe}
DisplayName          : HTTP(S) Inbound
Description          :
DisplayGroup        :
Group                :
Enabled              : True
Profile              : Domain, Private, Public
Platform             : {}
Direction            : Inbound
Action               : Allow
EdgeTraversalPolicy  : Block
LooseSourceMapping   : False
LocalOnlyMapping     : False
Owner                :
PrimaryStatus        : OK
Status               : The rule was parsed successfully from the store. (65536)
EnforcementStatus    : NotApplicable
PolicyStoreSource    : PersistentStore
PolicyStoreSourceType : Local

PS C:\Users\localadmin> _
```

We've now completed the server setup. We could configure a separate IIS site and app pool for our site. But, to keep things simple, we're going to use the default.

# Download Sample Website

## Overview

In this short step we will download our website - again, via PowerShell - into our default IIS directory.

## Download and Expand Site

The default folder for IIS is `C:\inetpub\wwwroot`. This is the folder we will use for hosting our site. Keep in mind that within a microservice architecture, a container has a single function or purpose. So, while we theoretically *could* host multiple sites in IIS, it's not best practice. Moving away from VMs will require us to rethink how we're accustomed to doing things.

We have a sample static site that we'll download from GitHub as a .zip file and expand it into our target folder.

1. If it's not still open from the previous step, open PowerShell with elevated privileges.
2. Type the following commands:

```
(new-object Net.WebClient).DownloadFile('https://github.com/AzureWorkshop/s/samples-simple-iis-website/archive/master.zip', 'D:\master.zip');  
Expand-Archive -LiteralPath D:\master.zip -Destination D:\  
(new-object -com shell.application).namespace('C:\inetpub\wwwroot\').Copy  
Here((new-object -com shell.application).namespace('D:\samples-simple-iis-  
-website-master').Items(), 16);  
del c:\inetpub\wwwroot\iisstart*.*
```

After a couple of seconds, the file should download. The above script downloads the .zip file from GitHub to the `D:\` temp drive (line 1); extracts the .zip file's contents (line 2) which, in-turn, creates a subdirectory of the extracted files; copies the files from the subdirectory to our IIS root folder (line 3); and, deletes the IIS placeholder files.

Now, if you were to open Internet Explorer on the server (e.g. `http://localhost/`), you should see our website that simply displays a 'Hello World' page.

# Construct Dockerfile

## Overview

Based on most of the previous steps, we are ready to build our Dockerfile. We will mimic those steps for automating our image construction.

## Review

In preparation of writing our Dockerfile, let's review all the steps we've performed up to this point.

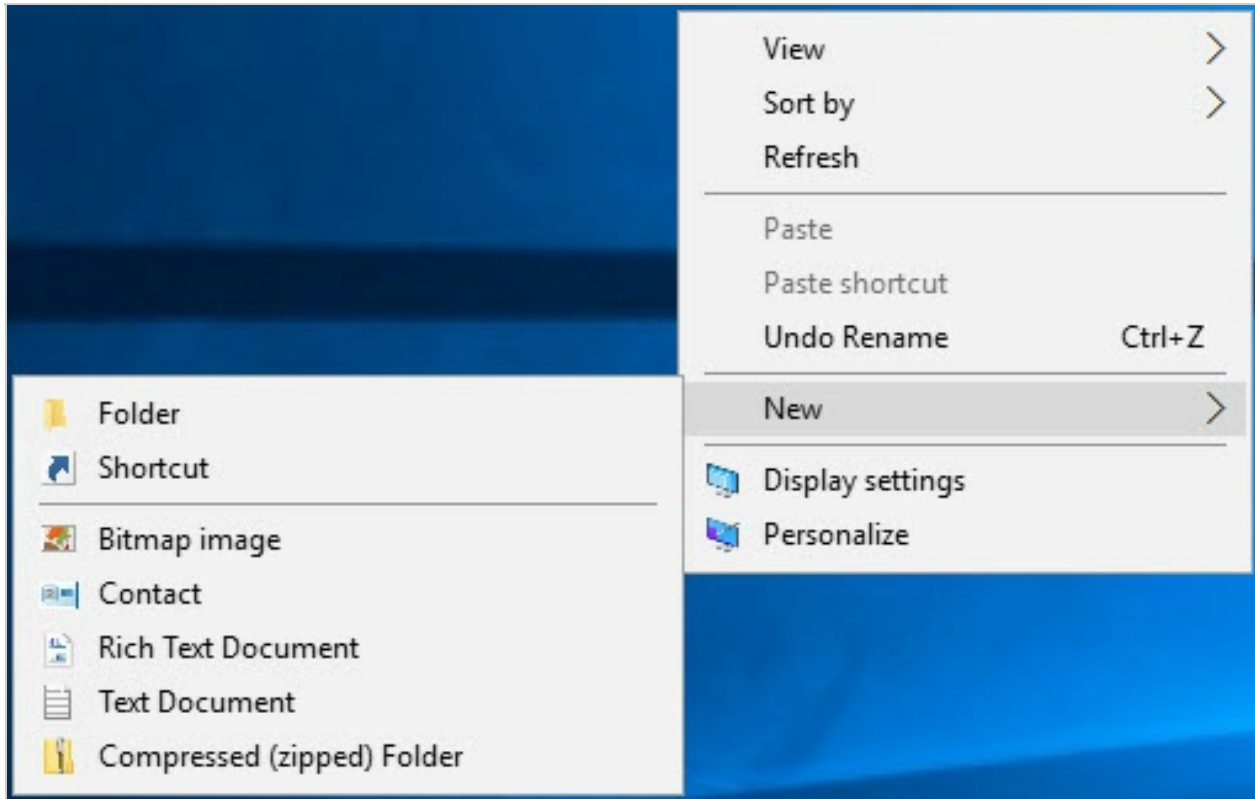
1. Install the latest version of Windows Server. Typically, we would use Windows Nano Server which is designed for containerized deployments. However, preparing a Nano Server container requires Hyper-V and is a little more in-depth than what we want to accomplish for this workshop, so we'll stick with Windows Server 2016.
2. Install the latest updates
3. Install and configure Docker
4. Install and configure IIS
5. Configure the firewall
6. Download the sample website

As a reminder, since we are constructing an image, we can **ignore** step 3. We won't need Docker installed inside of the image. Additionally, Microsoft is nice enough to provide us an image with IIS installed and the firewall configured. This allows us to **ignore** steps 4 and 5, as well. All we are required to do is install updates, Microsoft.NET and download our website.

## Create the Dockerfile

Let's go ahead and create the Dockerfile contents. We'll then examine each line below.

1. We need to create a Dockerfile. Somewhere on your desktop, **right-click**, then click **New**, followed by **Text Document**.



2. Name the new file **Dockerfile**. (Note: This will add the ".txt" extension to the file automatically. Typically, our Dockerfiles shouldn't have an extension, but that's okay. We'll work with it.)
3. Enter the following *without* the line numbers. The line numbers are provided for reference below.

```

1 FROM microsoft/iis:latest
2 SHELL ["powershell"]
3 MAINTAINER Your Name <you@yourcompany.com>
4
5 RUN Install-WindowsFeature NET-Framework-45-ASPNET ; \
6     Install-WindowsFeature Web-Asp-Net45 ; \
7     Install-WindowsFeature Web-App-Dev ; \
8     Install-WindowsFeature Web-Net-Ext45 ; \
9     Install-WindowsFeature Web-AppInit ; \
10    Install-WindowsFeature Web-ISAPI-Ext ; \
11    Install-WindowsFeature Web-ISAPI-Filter ;
12
13 RUN Invoke-Command -ScriptBlock {$sci = New-CimInstance -Namespace root/Microsoft/Windows/WindowsUpdate -ClassName MSFT_WUOperationsSession; Invoke-CimMethod -InputObject $sci -MethodName ApplyApplicableUpdates; exit }
14
15 RUN mkdir C:\temp
16
17 RUN (new-object Net.WebClient).DownloadFile('https://github.com/AzureWorkshops/samples-simple-iis-website/archive/master.zip', 'C:\temp\master.zip');
18 RUN Expand-Archive -LiteralPath C:\temp\master.zip -Destination C:\temp
19 RUN (new-object -com shell.application).namespace('C:\inetpub\wwwroot\').CopyHere((new-object -com shell.application).namespace('C:\temp\samples-simple-iis-website-master').Items(), 16);
20 RUN del c:\inetpub\wwwroot\iisstart*.*
21
22 EXPOSE 80

```

4. To save, **Ctrl+S**

## Explanation

First, if you remember from the previous steps, we were required to open a PowerShell prompt as an Administrator to allow the command to be executed with elevated privileges. By default, all Docker images execute under the identity of the built-in Administrator account.

**Line 1:** Specifies the base image, including the tag, with which we're starting. In our case, we are using the Microsoft Windows Server 2016 with IIS as the base image.

**Line 2:** Directs Docker to run everything from a PowerShell shell (not the default DOS/CMD prompt).

**Line 3:** Specifies the owner of the image with their email address.

**Lines 5-11:** Installs the Microsoft.NET Framework for ASP.NET and the ASP.NET extensions into IIS. Most of these should already be installed by default for the image we're downloading. However, this ensures that our system is up-to-date.

**Line 13:** Installs any necessary system updates. NOTE: On a *host* system or virtual machine, we would normally require a reboot. However, because we are simply building an image, the image will stop naturally once it's built. We will then only boot the image once we load it into a container.

Therefore, we theoretically have a built-in reboot in our process and a reboot here is not necessary.

**Line 15:** Creates a `temp` folder in which to store our `.zip` file. In our demo, we downloaded the `.zip` file to our temporary `D:\` drive. We don't have that drive in the container, so we'll use a temporary folder.

**Line 17:** Downloads the `.zip` file for our website to our `C:\temp` folder.

**Line 18:** Decompresses (expands) our `.zip` file into the `C:\temp` folder.

**Line 19:** When we decompress our `.zip` file, we create a subdirectory called `samples-simple-iis-website-master`. Here, we are copying the contents of that subfolder to our main IIS folder `C:\inetpub\wwwroot`.

**Line 20:** Deletes the two IIS placeholder files.

**Line 22:** IIS, by default, uses port 80. Therefore, similar to a firewall in the image, we open, or *expose*, the port to the outside host. We will bind to this open port later when we run a container based on this image. NOTE: This particular image, *microsoft/iis*, already exposes port 80 for us, so we're technically not required to add this line. However, it's still a good practice to explicitly include this line in case we need to reuse this Dockerfile or the underlying base ever changes.

That's it! That's all there is to creating a Dockerfile.



# Build Image

## Overview

Now that we have our Dockerfile, let's build our image from it.

## Build the Docker Image

Once we have our Dockerfile, building the image is pretty simple.

From the PowerShell window, type the following:

```
Get-Content "c:\users\localadmin\desktop\Dockerfile.txt" | docker build -t test /simpleweb -
```

This will build an image using `test/simpleweb` as the repository name. We are using PowerShell's `Get-Content` command to read the contents of our previously created Dockerfile and then pipe them to Docker's build command.

Due to the size of Windows Server, the initial build will take some time because it must download the base image first. Watch how Docker will step through our Dockerfile to build our image. Keep in mind while you watch this process that each step in our Dockerfile constitutes a layer in our image. We'll see the results of this below.

## Check Your Images

From the command prompt, type the following:

```
docker images
```

You should see something similar to:

REPOSITORY SIZE	TAG	IMAGE ID	CREATED
test/simpleweb 11.1GB	latest	9f4ec58ca830	3 minutes ago
microsoft/iis 10.6GB	latest	4f803ffceb53	37 hours ago

Our image has been built using the specified repository name. You'll also notice that the `microsoft/iis` image has been downloaded. This is because the build process required Windows Server with IIS in order to build our image. Now that our image has been built, you could delete the `microsoft/iis` image if you wanted to. Finally, when looking at the image sizes, you'll see that our image is 500MB larger due to the installation of Microsoft.NET, ASP.NET and other dependencies.

Be aware that the Nano Server image is only 1.07GB compared to the full Windows Server at 10.6GB which makes Nano Server more ideal for containers. Its really not best practice to use Windows Server in production as downloading 10.6GB and deploying that across your enterprise could consume a lot of bandwidth. For production, it's best to opt for Nano Server. But, again, Nano Server requires a little more preparation that extends a tad further beyond the scope of this workshop.

## View Image History

What if we wanted to see how our image is constructed? Or, what if we wanted to see exactly how much disk space each layer of our image required? We could find this out by checking the image's history.

```
docker image history test/simpleweb
```

When you run the above command, you see each command along from our Dockerfile along with it's layer id and the space requirements, if any.

We've now built a custom image based on a Dockerfile. We can use our custom image to deploy containers locally. Or, we could upload our image to a central repository so that others could leverage our image's functionality.

# Deploy Container

## Overview

Our custom image has now been created and is currently sitting in our local repository. Let's instantiate a container based on that image.

## Start a Container

To start a container from our image is very simple. The only thing we need to remember is exposing the internal port to the host.

```
docker run -d -p 8080:80 --name 'web_8080' test/simpleweb
docker run -d -p 8081:80 --name 'web_8081' test/simpleweb
docker run -d -p 8082:80 --name 'web_8082' test/simpleweb
```

We've started 3 separated instances of our web server. We've bound the web server's internal port 80 to three host ports (e.g 8080-8082). We've also supplied meaningful names to our containers. We can reference those containers by the names we've specified for easier management. For example, we can restart or stop a container using it's name instead of the container id.

Check the running images:

```
docker ps
```

You should see something like the following:

CONTAINER ID	IMAGE	COMMAND	CREATED
3d1929c8e1b5	test/simpleweb	"C:\\ServiceMonitor..."	3 seconds ago
Up 2 seconds	0.0.0.0:8082->80/tcp	web_8082	
323a65fa5143	test/simpleweb	"C:\\ServiceMonitor..."	11 seconds ago
Up 10 seconds	0.0.0.0:8081->80/tcp	web_8081	
7d4fee5c8f89	test/simpleweb	"C:\\ServiceMonitor..."	About a minute ago
Up 59 seconds	0.0.0.0:8080->80/tcp	web_8080	

Notice that all three containers are running, but, as we've specified, are bound to different ports and have custom names.

For practice, restart `web_8081` :

```
docker restart web_8081
```

Executing the command, may take a second. After it completes, check the running images again. You should now see that the uptime for `web_8081` is less than the other two containers.

We have now successfully created three container instances running our custom image.

## View the Container Websites

Before we attempt to expose our sites to the outside world, let's make sure that we can access them locally on the VM.

1. Open a web browser on the virtual server and try to navigate to `http://localhost:8080` (our `web_8080` container). Oops. It seems we received an error. What did we do wrong? Let's investigate.
2. Look again at the output of the `docker ps` command.
3. Notice the *Ports* column. Our external port is not mapped to the loopback address (e.g. `127.0.0.1` or `localhost`). Long story short, this is due to a way Windows maps its network interfaces.
4. We need to get the actual, virtual IP address of the container. To do this, type the following at the PowerShell prompt changing the container name for each running container:

```
docker inspect --format '{{ .NetworkSettings.Networks.nat.IPAddress }}' web_8080
```

5. Now, let's use the returned IP instead of the `localhost` to load our website. In the browser change the URL to `http://<web_8080's virtual IP address>:8080` (e.g. `http://172.26.67.126:8080`). This should display our *Hello World* sample web site.





# Expose Site in Azure

## Overview

The final part of this workshop is to practice exposing a container outside of Azure. We're going to create a simple web server and access it from our local machine. Due to the way Windows (specifically, Hyper-V) currently handles networking with Docker (e.g. the `0.0.0.0` IP assignment to our containers), this process is *much* simpler in Linux. So we will step through this slowly so that you understand the steps.

## Network Security Group (NSG)

When we created our Windows Server virtual machine, we accepted the defaults, including the default settings for our NSG. The default settings only allowed RDP (port 3389) access. We need to add a rule to our NSG to allow HTTP traffic over it's default port for our running web servers.

1. If you are not still there, go back to the Azure portal and navigate to the settings of your Windows Server virtual machine.
2. In the left menu, click on **Network interfaces**  .
3. This will open the *Network Interfaces* blade for your Windows Server virtual machine. Click on the singular, listed interface.
4. In the left menu, click on **Network security group**  .
5. This will list the currently active NSG. In our case, it should be the NSG that was created with our virtual machine - **docker-win-nsg**. Click on the NSG (**NOTE:** Click on the actual NSG link, **NOT** on **Edit**).
6. In the left menu, click on **Inbound security roles**  .
7. At the top of the blade, click **Add**  .
8. Enter the following configuration:
  - Service: **HTTP**
  - Port range: **80**
  - Priority: **100**
  - Name: **HTTP**

9. Click **OK**.

This will take a couple of seconds to complete.

## Docker Networking

Full disclosure, the Docker Networking topic is a very deep and complicated subject. There are many ways to accomplish this, especially, if you are using an orchestrator such as Docker Swarm or Kubernetes. We typically want a networking configuration that allows us to dynamically add containers (services) and have them auto-discovered. This is particularly critical for services that should autoscale based on demand.

For our workshop, we are going to sidestep this conversation and leave it to another workshop. Instead, we are going to create a network schema that will allow us to expose our individual containers *manually* via an IP address. In our case, we want a configuration that is similar to what's known as a *Host* mapping. By default, Docker on Windows only creates a NAT network. Therefore, we need to create our Host network manually. In Docker, this type of network configuration is known as *Transparent*.



For the next section, you will be switching back and forth between the Azure portal and your VM. Keep both open.

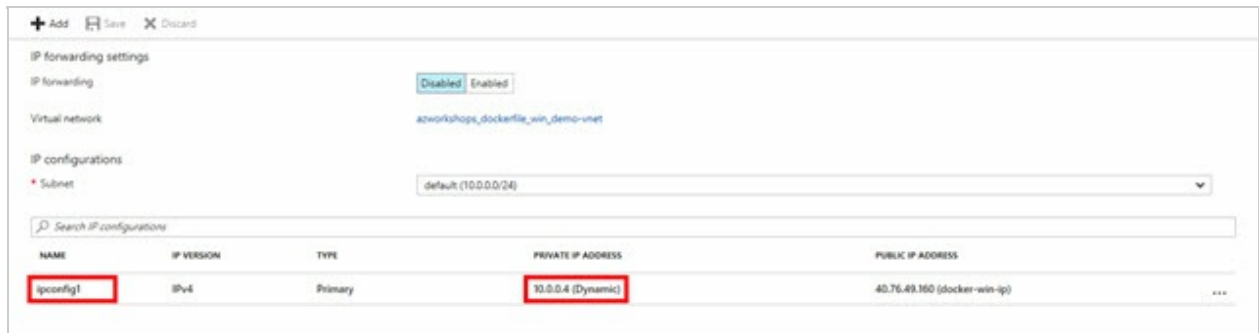
### Set Server Static IP

When we created our VM, we opt'ed that the virtual machine's *internal* IP was set dynamically. We now want to set it to a static internal IP; and, we have to do this in two places - Azure portal and the VM, itself.

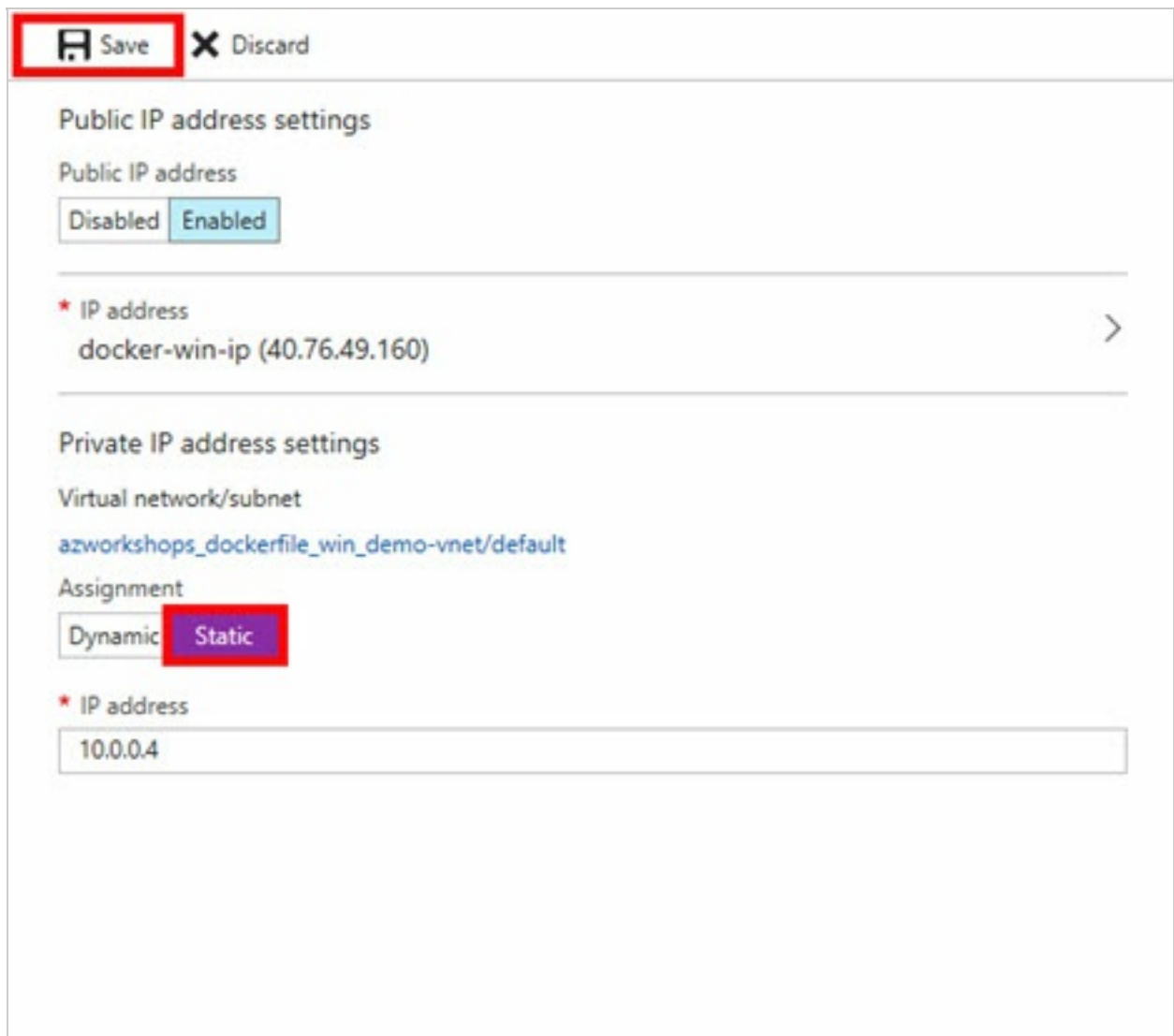
#### Static IP - Azure

Let's begin by setting the static IP in Azure.

1. If you are not still there, go back to the Azure portal and navigate to the settings of your Windows Server virtual machine.
2. In the left menu, click on **Network interfaces**  .
3. This will open the *Network Interfaces* blade for your Windows Server virtual machine. Click on the singular, listed interface.
4. In the left menu, click on **IP configurations**  .
5. This will list all of the VM's currently assigned IPs. At the moment, there should only be one IP configuration listed **ipconfig1**. Under the table heading *PRIVATE IP ADDRESS*, we should see that it is assigned dynamically. Click on **ipconfig1**.



6. Approximately 80% down on the resulting blade, you should see a toggle between *Dynamic* and *Static*. Click on **Static**.



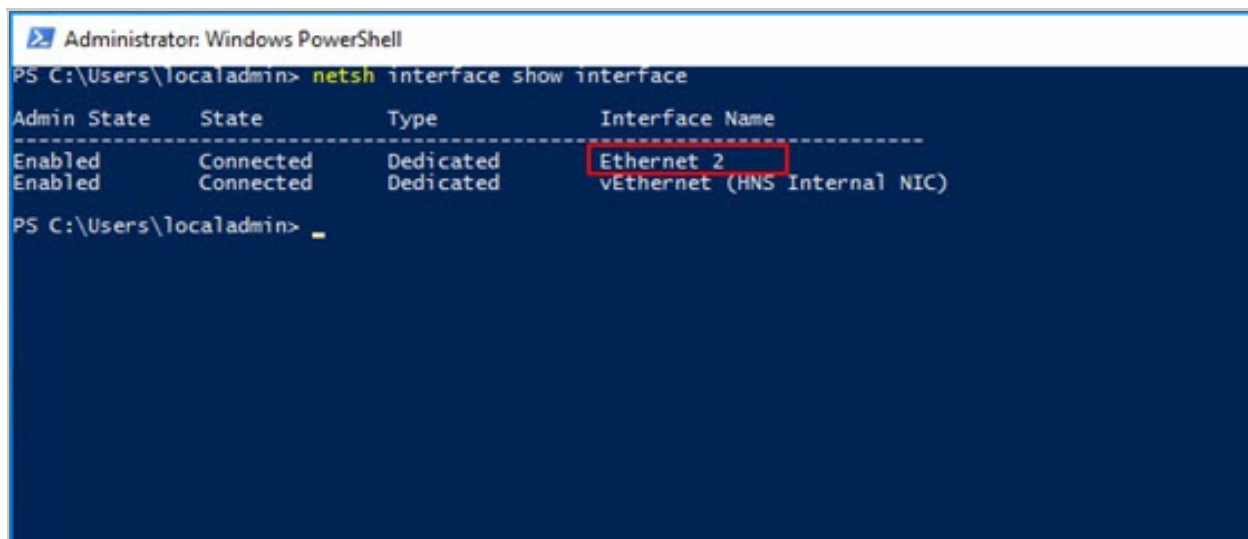
7. This should enable the input box for the IP address. Leave it as-is.
8. At the top of the blade, click **Save**.

This will take a second, but after the save has been completed, the *Static* button should turn from purple to light blue and the *Save* button should be disabled.

## Static IP - VM

Now, let's set the static IP on the virtual machine.

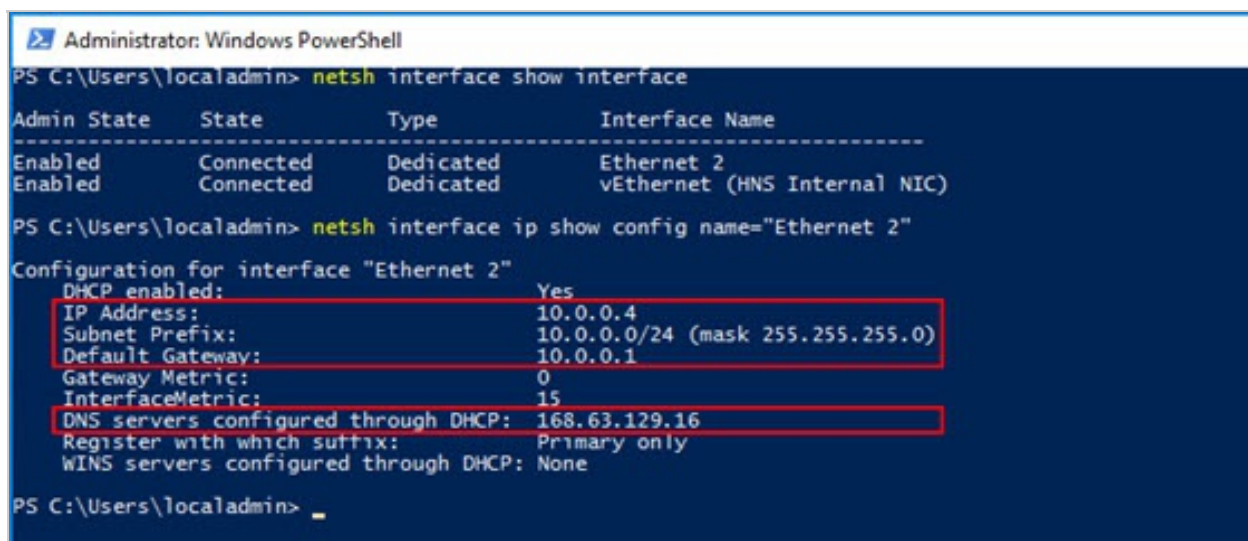
1. On the VM, at the PowerShell prompt (as Administrator), type `netsh interface show interface`. You will see something like the following table:



```
Administrator: Windows PowerShell
PS C:\Users\localadmin> netsh interface show interface
Admin State      State            Type             Interface Name
-----
Enabled          Connected       Dedicated        Ethernet 2
Enabled          Connected       Dedicated        vEthernet (HNS Internal NIC)
PS C:\Users\localadmin> _
```

The *vEthernet* is a virtual adapter added by Docker. It's the internal, NAT adapter. We want the *Ethernet* adapter. In our case, it's **Ethernet 2**, but it could be 1, 3 or some other number. It's our primary adapter provided to us by Hyper-V.

2. Again, at the PowerShell prompt, type `netsh interface ip show config name="Ethernet 2"` (obviously, substitute "Ethernet 2" for the name of your ethernet adapter if it is different). **IMPORTANT:** Keep this information handy as you will need it for a couple of steps below.

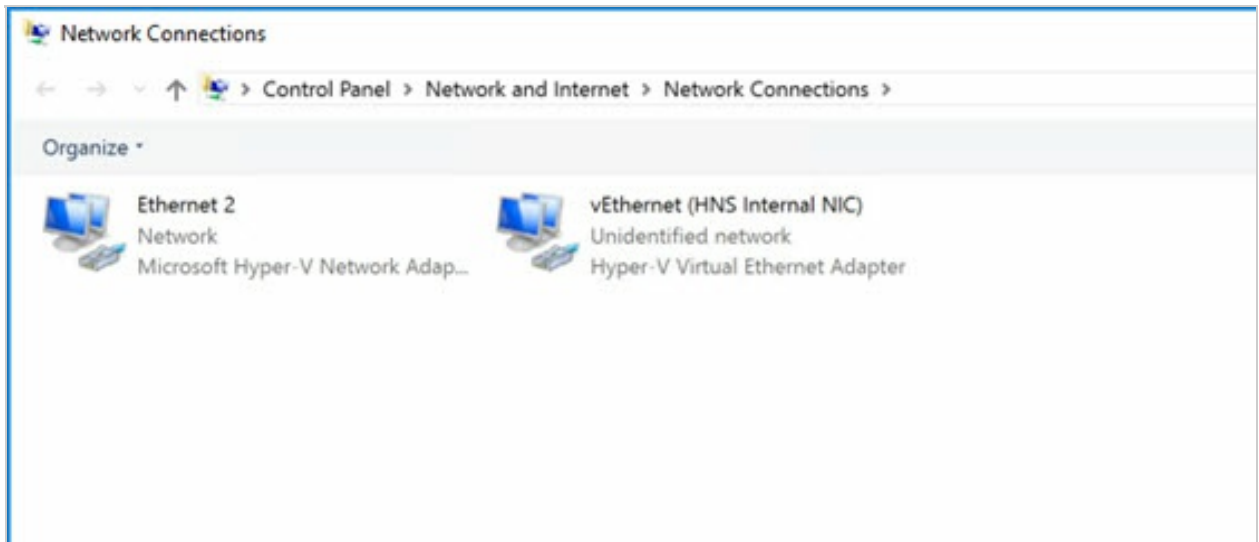


```
Administrator: Windows PowerShell
PS C:\Users\localadmin> netsh interface show interface
Admin State      State            Type             Interface Name
-----
Enabled          Connected       Dedicated        Ethernet 2
Enabled          Connected       Dedicated        vEthernet (HNS Internal NIC)
PS C:\Users\localadmin> netsh interface ip show config name="Ethernet 2"
Configuration for interface "Ethernet 2"
DHCP enabled:    Yes
IP Address:      10.0.0.4
Subnet Prefix:   10.0.0.0/24 (mask 255.255.255.0)
Default Gateway: 10.0.0.1
Gateway Metric: 0
InterfaceMetric: 15
DNS servers configured through DHCP: 168.63.129.16
Register with which suffix: Primary only
WINS servers configured through DHCP: None
PS C:\Users\localadmin> _
```

This will show us the necessary configuration (outlined with a red border) to manually configure our adapter settings.

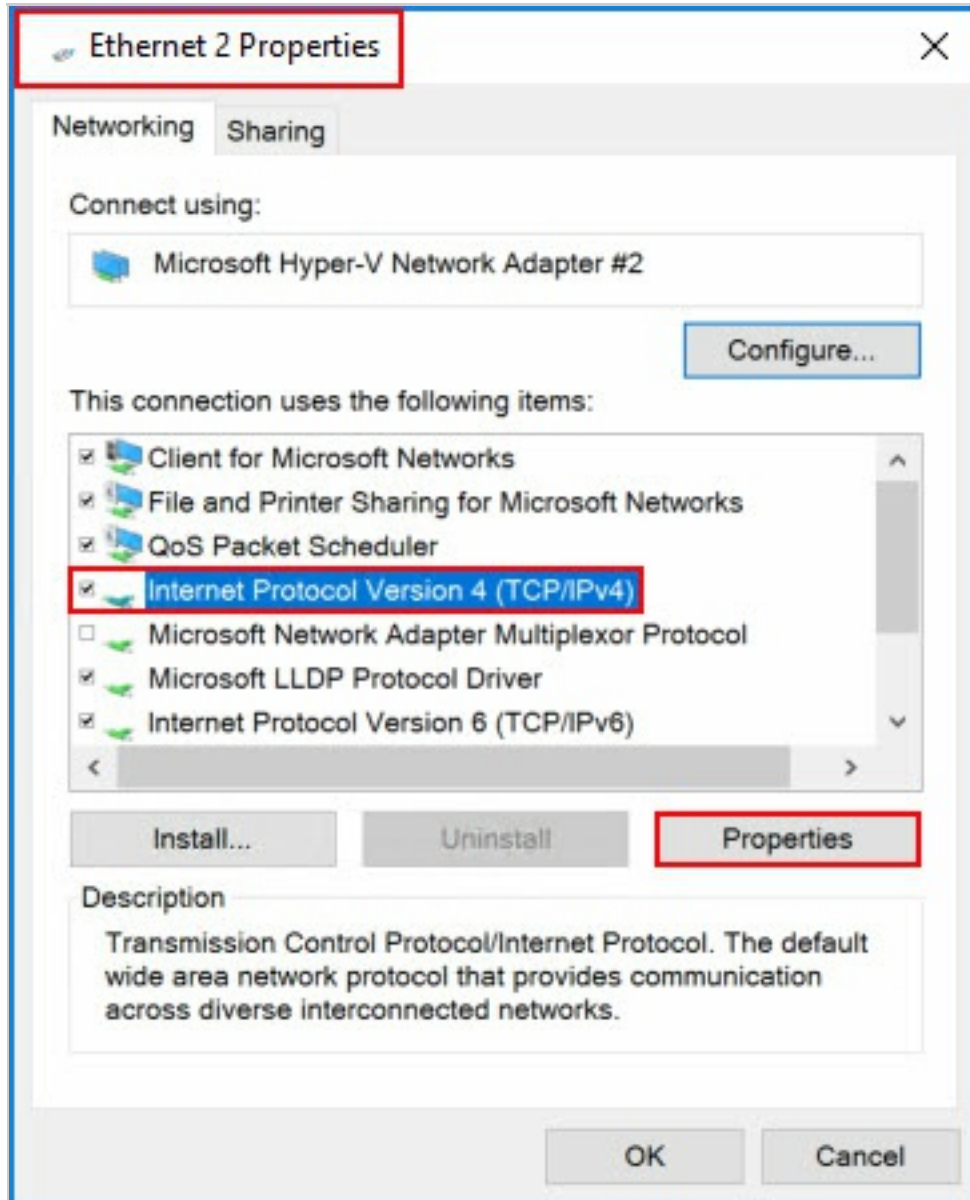


3. At the PowerShell prompt, type `ncpa.cpl` . This will show something like the following.

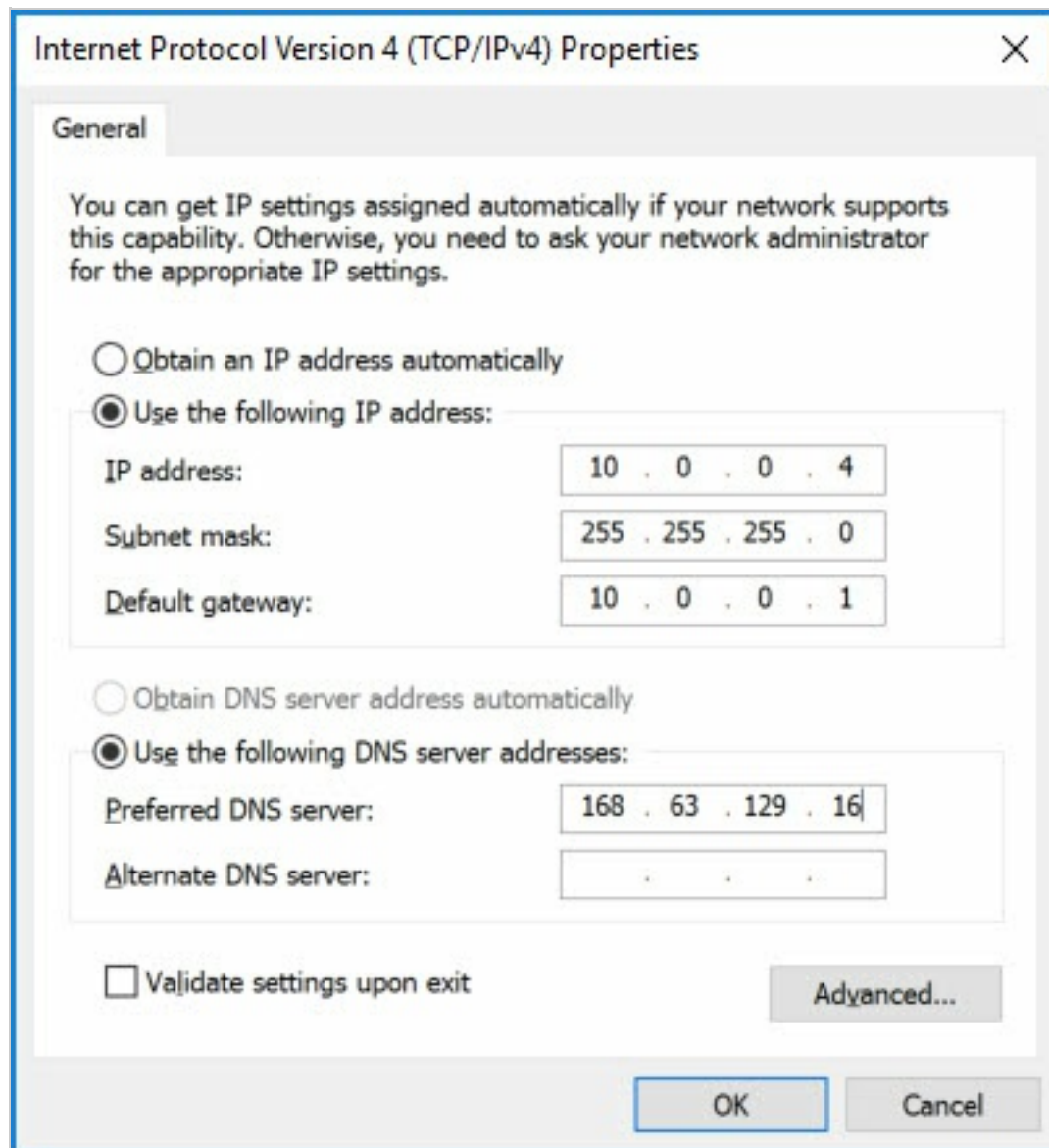


4. Right-click on your primary ethernet adapter (e.g. "Ethernet 2"). Select **Properties**.

5. In the dialog window, select **Internet Protocol Version 4 (TCP/IPv4)** and click on **Properties**.



6. In this dialog, select **Use the following IP address:** and **Use the following DNS server addresses:**. Additionally, enter the information you acquired from the previous PowerShell `netsh` command. Your information may not be exactly like below (again, compare the information to the `netsh` output), but it should be similar.



7. Click **OK**.

8. Click **Close**. (NOTE: You may *temporarily* lose your connection, but you should automatically be reconnected within a few seconds.)

## Create a Transparent Network

We are now ready to create a *Transparent*, or Host, network within Docker that will allow us to create containers that have IP addresses on the parent virtual machine's subnet. This will provide a direct route from inside or outside of our network to the container.

Let's start by examining the current networks Docker has created for us.

On the VM, at the PowerShell prompt, type `docker network ls`. This should return something similar to the following:

NETWORK ID	NAME	DRIVER	SCOPE
7d22076d85e0	nat	nat	local
3112bd939814	none	null	local

You will see two networks listed - `nat` and `none` . Anything attached to the `none` network is not accessible from a network. The `nat` network is what our current three containers are attached to. It is what allows us to connect to them from the virtual machine's web browser. The `nat` network is also what allows the containers to communicate with each other. It's, basically, a network that's *internal* to that virtual machine.

To view more information and see the containers currently connected to the network, type `docker network inspect nat` .

Among other things, you'll see the subnet, gateway and the containers attached to the network.

In this process, we are going to eventually *reserve* some IP addresses within our subnet to be used by our containers. Again, keep in mind, that our containers must reside on the *same subnet* as our VM.

### Multi-Homing

Ideally, you would probably have two NICs attached to this VM and set it up as a multi-homed server. This would allow you to have a separate subnet that's dedicated to your containers. However, this is not the recommended setup for production as you would utilize some type of orchestrator with service discovery and a mesh network.

Let's create our transparent network to sit inside our subnet.

In PowerShell, type the following and press Enter.

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

**IMPORTANT:** Refer to the information you received from the last `netsh` command. (NOTE: You may *temporarily* lose your connection, but you should automatically be reconnected within a few seconds.)

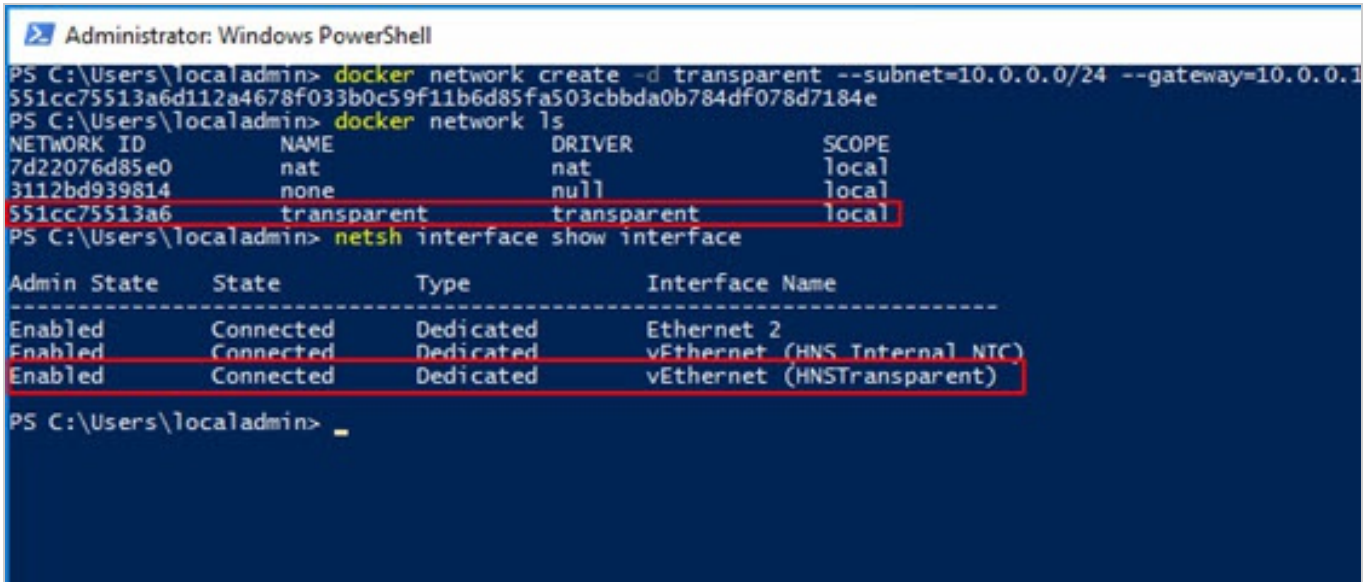
This command tells Docker to create a new network using the *transparent* driver ( `-d` ) with a `10.0.0.0/24` subnet and `10.0.0.1` gateway, naming it "transparent".

Now, again, at the PowerShell prompt, type `docker network ls` . You should now see the network listed.

Additionally, type in:

```
netsh interface show interface
```

This will allow you to view the available network interfaces (NICs).



```
Administrator: Windows PowerShell
PS C:\Users\localadmin> docker network create -d transparent --subnet=10.0.0.0/24 --gateway=10.0.0.1
551cc75513a6d112a4678f033b0c59f11b6d85fa503cbbda0b784df078d7184e
PS C:\Users\localadmin> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
7d22076d85e0       nat                 nat                 local
3112bd939814       none               null                local
551cc75513a6       transparent        transparent         local
PS C:\Users\localadmin> netsh interface show interface

Admin State   State           Type             Interface Name
-----
Enabled       Connected       Dedicated        Ethernet 2
Enabled       Connected       Dedicated        vEthernet (HNS Internal NIC)
Enabled       Connected       Dedicated        vEthernet (HNSTransparent)

PS C:\Users\localadmin>
```

You'll notice that, when we created our transparent network, Docker created a new virtual network interface *vEthernet (HNSTransparent)*. This is the NIC we'll add our IPs to in the steps below.

We've now created our transparent network and we're ready to add our publicly accessible containers.

## Add Public Containers

After all of that, we're now finally ready to add our public containers and access them from outside of Azure.

**From this point onward, each step below should be repeated for *each* container we wish to add. Simply change the IP address.**




### Pick a "Reserved" IP

Theoretically, our transparent network doesn't exist outside of our VM so Azure DHCP/DNS will not automatically assign an IP address to our container. We must assign an IP address to it manually. First, let's pick an IP.

I'm going to start with \*.\*.\*.100 as my first container's IP address. For my network (your's may be different), the full IP address will be 10.0.0.100 with a subnet mask of 255.255.255.0 (again, you get the subnet and mask from the `netsh` command).

### "Reserve" the IP in Azure

So, first, we're not *really* reserving the IP address. But, in a way, we kind of are. We are going to manually assign our IP as a *secondary* IP to our virtual machine's NIC.

1. If you are not still there, go back to the Azure portal and navigate to the settings of your Windows Server virtual machine.
2. In the left menu, click on **Network interfaces**  .
3. This will open the *Network Interfaces* blade for your Windows Server virtual machine. Click on the singular, listed interface.
4. In the left menu, click on **IP configurations**  . This time we'll assign a secondary IP address.
5. In the top menu, click on **Add**  .
6. In the fields make the following selections:
  - o Name: **web\_public100**
  - o Allocation: **Static**
  - o IP address: **10.0.0.100** (use the IP address you chose above)
  - o Public IP address: **Enabled**
  - o IP address: (click on it & **Create New**)
    - Name: **web\_public100**
    - Assignment: **Dynamic**
7. Click **OK**.

This will take a second; be patient. Once it has completed, you'll see the new IP configuration listed in the table. Make note of the **PUBLIC IP ADDRESS** for the new IP configuration. You will use this IP address to access your container from a web browser once we're done.

We've added the necessary configuration in Azure to route requests for that IP to our VM. In the next step, we'll add the IP to the machine so that it will *listen* on that IP address.

## Assign the IP to the Virtual NIC

Remember the new virtual NIC that was created above when we created our transparent network? We need to add an IP address to it. Most of the time you would add the IP address through the GUI, but we cannot do this. We need to use a feature called "SkipAsSource" ([more info](#)). Therefore, we must use the `netsh` command once again so that the SkipAsSource feature is not enabled.

Go back to your VM and at the command prompt, type:

```
netsh int ipv4 add address "vEthernet (HNSTransparent)" 10.0.0.100 255.255.255.0
```

(again, use the virtual NIC name, IP address and subnet mask acquired from the various steps above)

Now, type the command

```
netsh interface ip show config name="vEthernet (HNSTransparent)"
```

We see that both IPs (our original static IP and our new IP) have been assigned to our transparent network:

```
Administrator: Windows PowerShell
PS C:\Users\localadmin> netsh interface ip show config name="vEthernet (HNSTransparent)"
Configuration for interface "vEthernet (HNSTransparent)"
DHCP enabled: No
IP Address: 10.0.0.4
Subnet Prefix: 10.0.0.0/24 (mask 255.255.255.0)
IP Address: 10.0.0.100
Subnet Prefix: 10.0.0.0/24 (mask 255.255.255.0)
Default Gateway: 10.0.0.1
Gateway Metric: 256
InterfaceMetric: 15
Statically Configured DNS Servers: 168.63.129.16
Register with which suffix: Primary only
Statically Configured WINS Servers: None
PS C:\Users\localadmin> _
```

Our machine will now listen for requests on that IP address. Our routing is complete. Now, we simply need to add a container and assign it that IP address.

**IMPORTANT:** Our transparent network is a virtual network on our machine. If we delete our transparent network in Docker, then we also remove the IPs associated with our virtual NIC. Just be aware.

## Create a Container on Our Transparent Network

We already have our first three containers running on our `nat` network. Unfortunately, there's no easy way to reconfigure a container's port mapping; and, you can't change it while the container is running. So, we'll create a new container and attach it to our transparent network.

From the PowerShell command line, type

```
docker run -d --net=transparent --ip=10.0.0.100 --name "web_public100" test/simpleweb
```

This command does a couple of things. First, as you may remember from earlier, we are running this container in the background, or as "detached" ( `-d` ). Second, we explicitly specify the network in which to attach our container. In our case `transparent` . When we use a transparent network (or a few other types), we're required to specify the IP address as, again, the host network's DHCP cannot assign an IP address. The rest of this command should be familiar.

Let's view our running containers by typing `docker ps` .

CONTAINER ID	IMAGE	COMMAND	CREATED
821e2dc235d4	test/simpleweb	"C:\\ServiceMonitor..."	18 minutes ago
o	Up 16 minutes	80/tcp	web_public100
1c84a5399eaa	test/simpleweb	"C:\\ServiceMonitor..."	3 hours ago
	Up 3 hours	0.0.0.0:8082->80/tcp	web_8082
3b644253ff84	test/simpleweb	"C:\\ServiceMonitor..."	3 hours ago
	Up 3 hours	0.0.0.0:8081->80/tcp	web_8081
82dc9c21c5f2	test/simpleweb	"C:\\ServiceMonitor..."	3 hours ago
	Up 3 hours	0.0.0.0:8080->80/tcp	web_8080

We now see our `web_public100` running, but there is no NAT translation - it's simply listening on port 80. If we *inspect* the container's configuration (e.g. `docker container inspect web_public100`), we see our assigned IP address closer to the end of the output.

Whew! That's it! We've added a Widows container and made it accessible outside of Azure. Let's test our work.

## Test on the VM

Let's make sure we can access the container from our VM.

1. On the VM, open up Internet Explorer.
2. In the URL, type the IP address, including 'http://' (e.g. `http://10.0.0.100`). NOTE: We don't have to use a port this time as the container is mapped directly to port 80.

If successful, you should see the 'Hello World' web page.

## Test Outside of Azure

Now that we know our container is accessible from a IP address from within our subnet, let's make sure it's accessible from outside of Azure.

1. On your *local* machine, open a web browser.
2. Use the *public IP address* you acquired from adding the "Reserved IP" to your machine and type it (including the 'http://') into the URL.

Success! Again, your should see the 'Hello World' web page.

## Review

Gee, that was a bit of work! As I stated earlier, doing this in Linux is much easier. If you haven't done so already [try it out](#). Since this was quite a bit of effort, I wanted to quickly review what we've done.

1. We opened up port 80 in Azure's firewall to allow HTTP traffic to flow through.
2. We changed our IP configuration for the VM to a static IP from a dynamic IP so that we could later add additional IPs to the VM's NIC and not mess up routing should the VM reboot.



3. We created a *transparent* network in Docker to allow our containers to connect directly to our Host subnet.
4. We picked a "reserved IP" and then:
  1. Added that IP to our VM in Azure as a new, static IP configuration
  2. Added that IP to our virtual NIC created by the Docker transparent network
  3. Created a new container and assigned it to our transparent network and assigned it the IP

All of the steps on this page, simply comes down to the previous 4-7 steps. Also, remember, now that the networking is setup (steps 1-3), you only need to follow step 4 for all future containers on this VM that should have external access.

## Next Steps

As stated multiple times, this is not the ideal scenario when you need a fully-scalable and redundant solution. For those types of environments, it is recommended that you use an orchestrator like Docker Swarm or Kubernetes.

However, with that said, you could run step 4 again and add another web server container to your VM's Docker. Now that you have two containers on the subnet, you could add a load balancer in Azure for a bit of redundancy. Of course, it's only as redundant as the VM, itself. For this, you would probably want to add an Availability Set with multiple VMs hosting Docker. Then, load balance across the multiple Docker containers.