Docker on Ubuntu

Friday, August 4, 2017 1:48 PM

Install and configure Docker, along with deploying and managing Linux-based containers, on an Ubuntu server.

This is a short workshop to introduce you to Linux-based containers. In this workshop, you will gain experience in installing and configuring Docker on an Ubuntu server. You'll then deploy a couple of different images as containers to the server and experiment with managing those images and containers. Finally, you will configure Azure to allow you to access those containers from outside of your virtual network.

What You Will Learn

- Installing and Configuring Docker on Ubuntu
- Downloading and Managing Images
- Deploying and Working With Containers
- Exposing Docker Services in Azure

Ideal Audience

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

Overview

This is a short workshop to introduce you to Linux-based containers. In this workshop, you will gain experience in installing and configuring Docker on an Ubuntu server. You'll then deploy a couple of different images as containers to the server and experiment with managing those images and containers. Finally, you will configure Azure to allow you to access those containers from outside of your virtual network.

Time Estimate: 2.5 hours

Requirements

Setup Requirements

The following workshop will require that you use a Telnet/SSH client in order to connect to a remote machine. If you do not have a SSH client, then PuTTY will work fine. Depending on your environment, download the executable in a standalone file (.EXE) or an installable package (.MSI), either in a 32-bit or 64-bit.

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.

Exploring Azure

Objective

The first objective is for you to become familiar with connecting to and navigating the Azure portal. This will not be a difficult exercise, but will nonetheless demonstrate how to work within the Azure user interface.

Azure Portal Basics

Let's start by connecting to the Azure portal and becoming familiar with navigation.

1. Open a browser and navigate to http://www.azure.com.

2. In the top-right corner of your screen, you will see the menu option **PORTAL**. Click on it.

3. If you have not already, you will be required to authenticate.

4. After authentication is successful, you will be directed to your *Dashboard*. The dashboard is configurable by adding, removing and resizing *tiles*. Additionally, you can have multiple dashboards depending on your preferences. You could have different dashboards for resources dedicated to different functions, lines of business, or for operations.

5. On the left will be your primary navigational menu. You should see a list of favorited services on the menu with descriptions. (NOTE: The number of options listed in your menu may differ from that of others depending on the number of services you have selected as a favorite.) If all you see are icons (no descriptions) on your menu, your menu is currently collapsed. Click the "hamburger" ☰ to expand it.

6. Pretty close to the top of your menu, you should see **Resource Groups** . Click this option.

7. Upon clicking the Resource Groups menu item, a *blade* will open revealing any created resource groups. In order to create resources in Azure, you must assign/place it in a resource group.

This is where we will get started creating our resources.

While this introduction wasn't too technical, it is sufficient for getting us to a point where we can begin the specifics in the workshop. If you'd like to look around a bit more, click a few of the other options in the main menu. Then, when you are ready, can you proceed to the next step.

Create a Virtual Machine

Objective

Now that we've explored the Azure portal a bit, let's get started with creating some resources. Our primary resource will be a virtual machine on which we install Docker. Once we create the virtual machine, we'll see that some additional resources are created for us.

Create a Resource Group

As stated on the previous page, 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_docker_ubuntu_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.

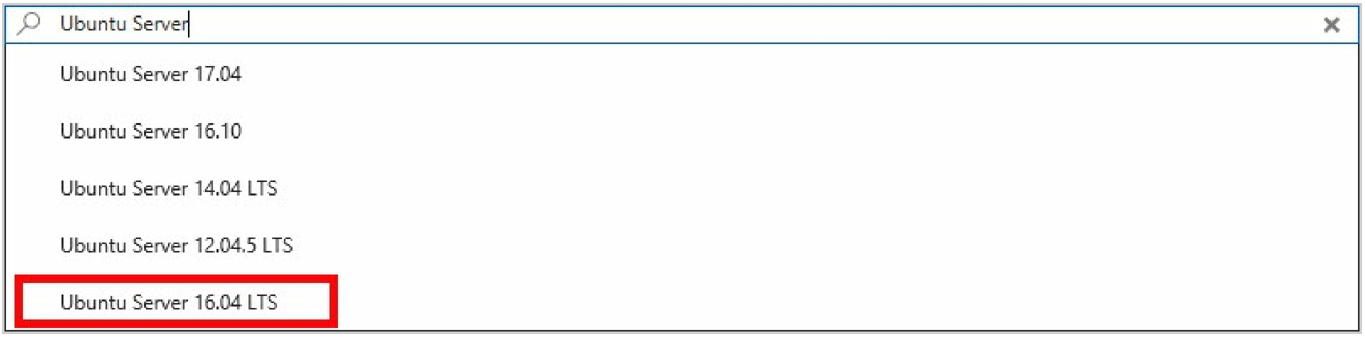
Create a Virtual Machine

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

1. If you are not there already, go ahead and navigate to the **azworkshops_docker_ubuntu_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 an Ubuntu server. Therefore, in the *Search Everything* box, type in **Ubuntu Server**. This will display a couple of different versions. Since we want to deploy the latest *stable* version of Ubuntu, from the displayed options, choose **Ubuntu Server 16.04 LTS**.



Ubuntu Server
Ubuntu Server 17.04
Ubuntu Server 16.10
Ubuntu Server 14.04 LTS
Ubuntu Server 12.04.5 LTS
Ubuntu Server 16.04 LTS

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

5. 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-ubuntu**
        - VM disk type: **SSD**
        - Username: **localadmin**
        - Authentication type: **Password**
          (NOTE: You can choose SSH if you are familiar with how to set this up. If you are not, we will do this in a later workshop. However, for this workshop, *Password* authentication is sufficient.)
        - Password: **Pass@word1234**
        - Confirm password: *<same as above>*
        - Subscription: *<choose your subscription>*
        - Resource group: **Use existing** - **azworkshops_docker_ubuntu_demo**
        - Location: *<choose a location>*

    2. Size

        - **DS1_V2**

    3. Settings

        - Use managed disks: **No**

        - Storage account: (click on it & **Create New**)

        - Name: **dockerubuntudata**<*random number*> (ex. *dockerubuntudata123456*)
          (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_docker_ubuntu_demo-vnet*)
    - Subnet: <*accept default*> (e.g. *default (172.16.1.0/24)*)
    - Public IP address: <*accept default*> (e.g. *(new) docker-ubuntu-ip*)
    - Network security group (firewall): <*accept default*> (e.g. *(new) docker-ubuntu-nsg*)
    - Extensions: **No extensions**
    - Availability set: **None**
    - Boot diagnostics: **Enabled**
    - Guest OS diagnostics: **Disabled**
    - Diagnostics storage account: (click on it & **Create New**)
        - Name: **dockerubuntudiags**<*random number*> (ex. *dockerubuntudiags123456*)
        - Performance: **Standard**
        - Replication: **Locally-redundant storage (LRS)**
4. Summary (just click **OK** to continue)

This machine is relatively small, but with containers, it can still deliver some pretty impressive performance. 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.

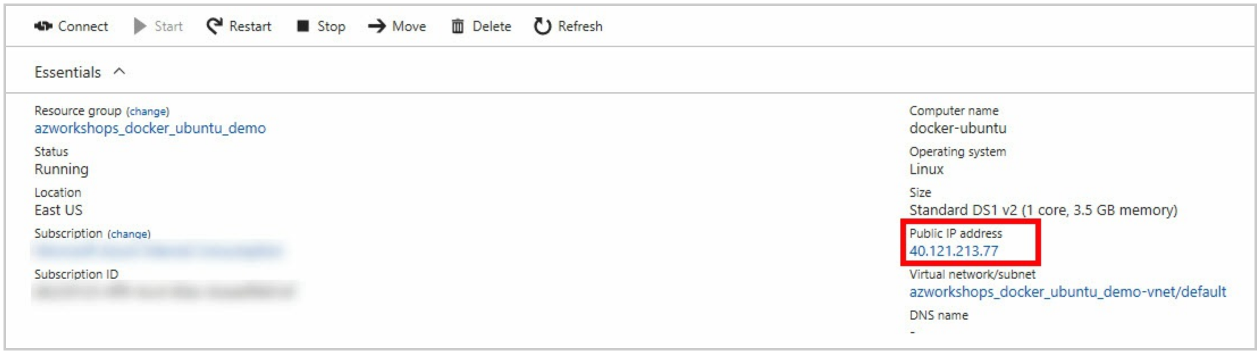
Connect to the Virtual Machine

Once your machine has been created, we can remotely connect to it using secure shell (SSH). These instructions assume that you do not have strong familiarity with SSH and/or that you have no built-in SSH client in your local OS. For this reason, we will be using the PuTTY client we downloaded earlier for the workshop. However, if you are more comfortable using another Telnet/SSH client (e.g. MacOS, Linux, Windows Sub-Layer (WSL)), please feel free to use it.

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.



Resource group (change)
azworkshops_docker_ubuntu_demo

Status
Running

Location
East US

Subscription (change)
[redacted]

Subscription ID
[redacted]

Computer name
docker-ubuntu

Operating system
Linux

Size
Standard DS1 v2 (1 core, 3.5 GB memory)

Public IP address
40.121.213.77

Virtual network/subnet
azworkshops_docker_ubuntu_demo-vnet/default

DNS name
-

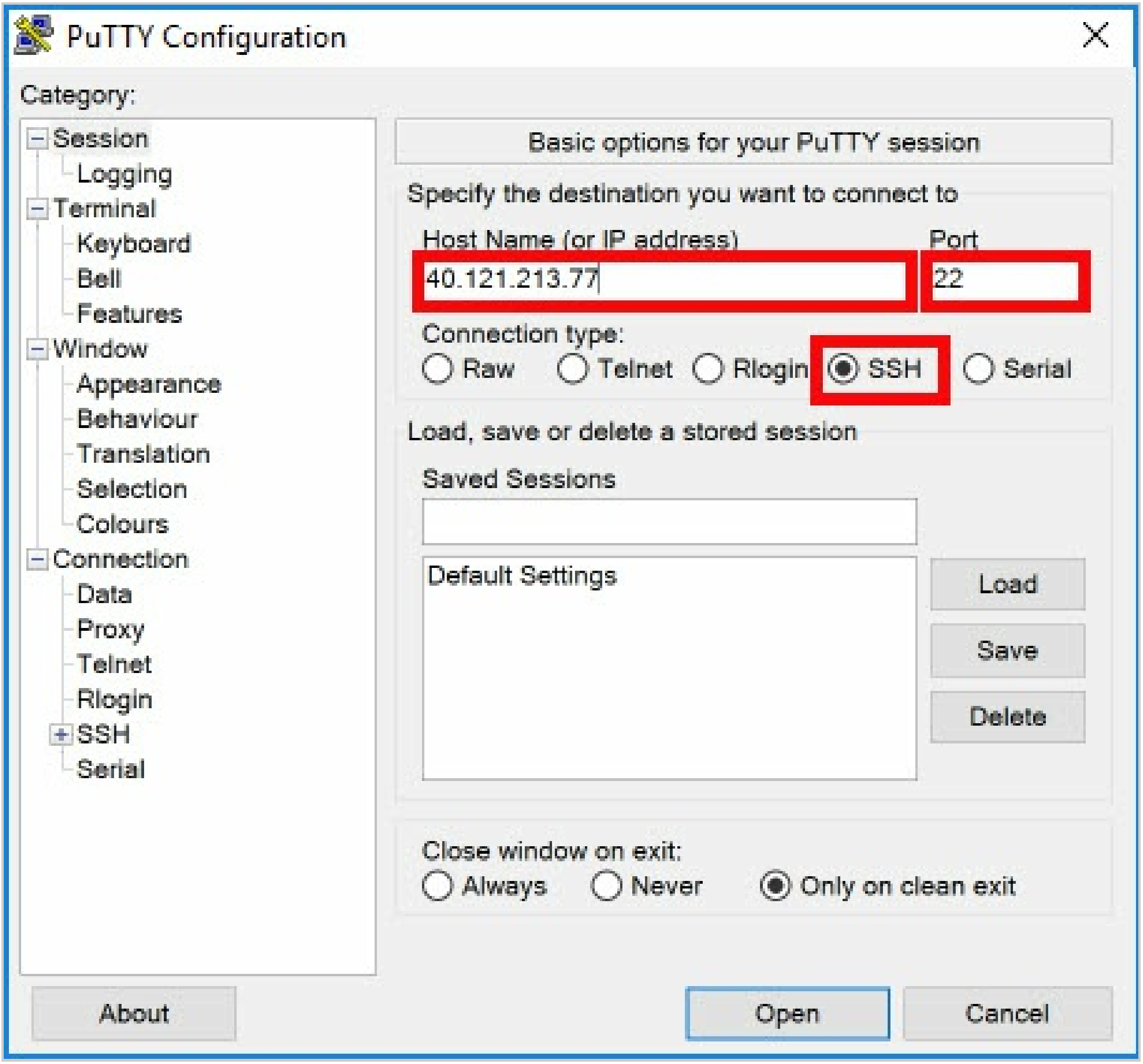
3. Copy the IP address.

Connect with SSH

1. Open **PuTTY**.

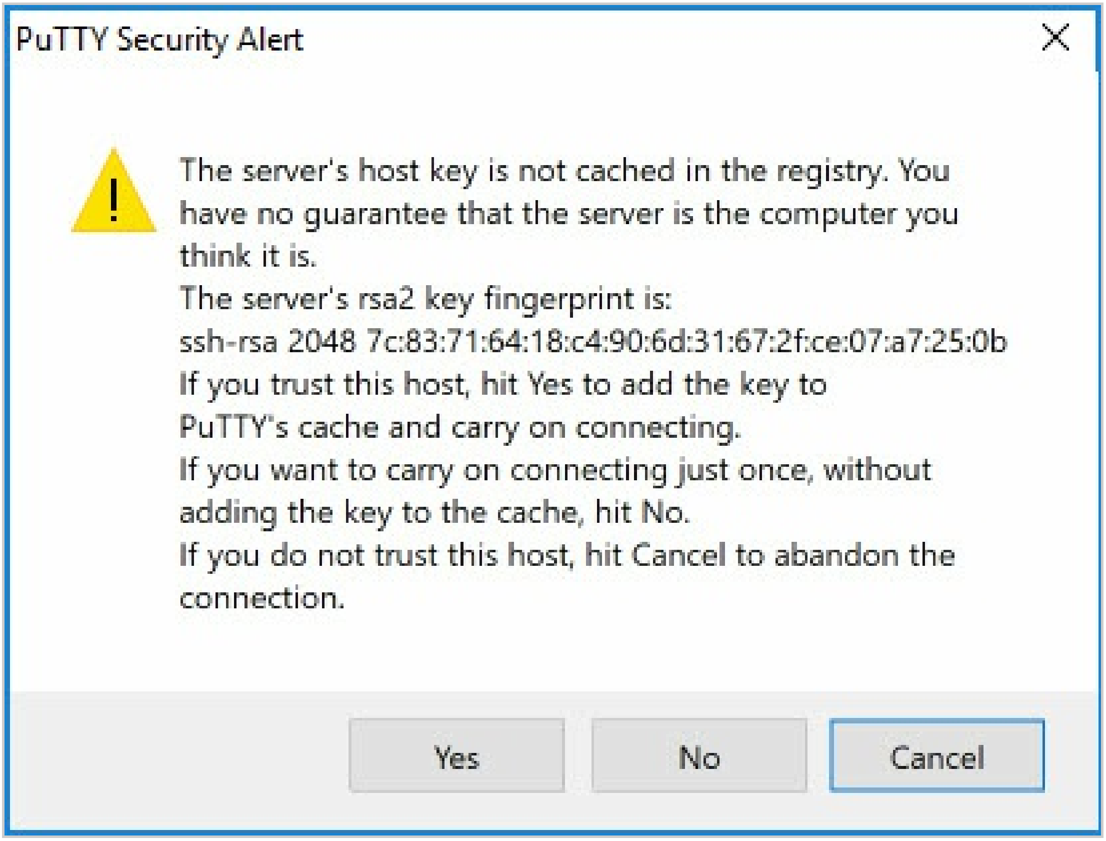
2. In the configuration window:

   - Hostname: ***<IP address from previous step>***
   - Port: **22**
   - Connection type: **SSH**



Host Name (or IP address): 40.121.213.77
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Close window on exit: Always Never [✓ Only on clean exit]

3. Click **Open**
4. In the security prompt, click **Yes**.


PuTTY Security Alert

The server's host key is not cached in the registry. You have no guarantee that the server is the computer you think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 2048 7c:83:71:64:18:c4:90:6d:31:67:2f:ce:07:a7:25:0b
If you trust this host, hit Yes to add the key to PuTTY's cache and carry on connecting.
If you want to carry on connecting just once, without adding the key to the cache, hit No.
If you do not trust this host, hit Cancel to abandon the connection.

5. You will then connect to the remote Ubuntu server.
6. Enter the username and password from above (e.g. **localadmin** and **Pass@word1234**, respectively).
7. You should then see the `bash` prompt:

```
localadmin@docker-ubuntu:~$
```

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

Install Docker

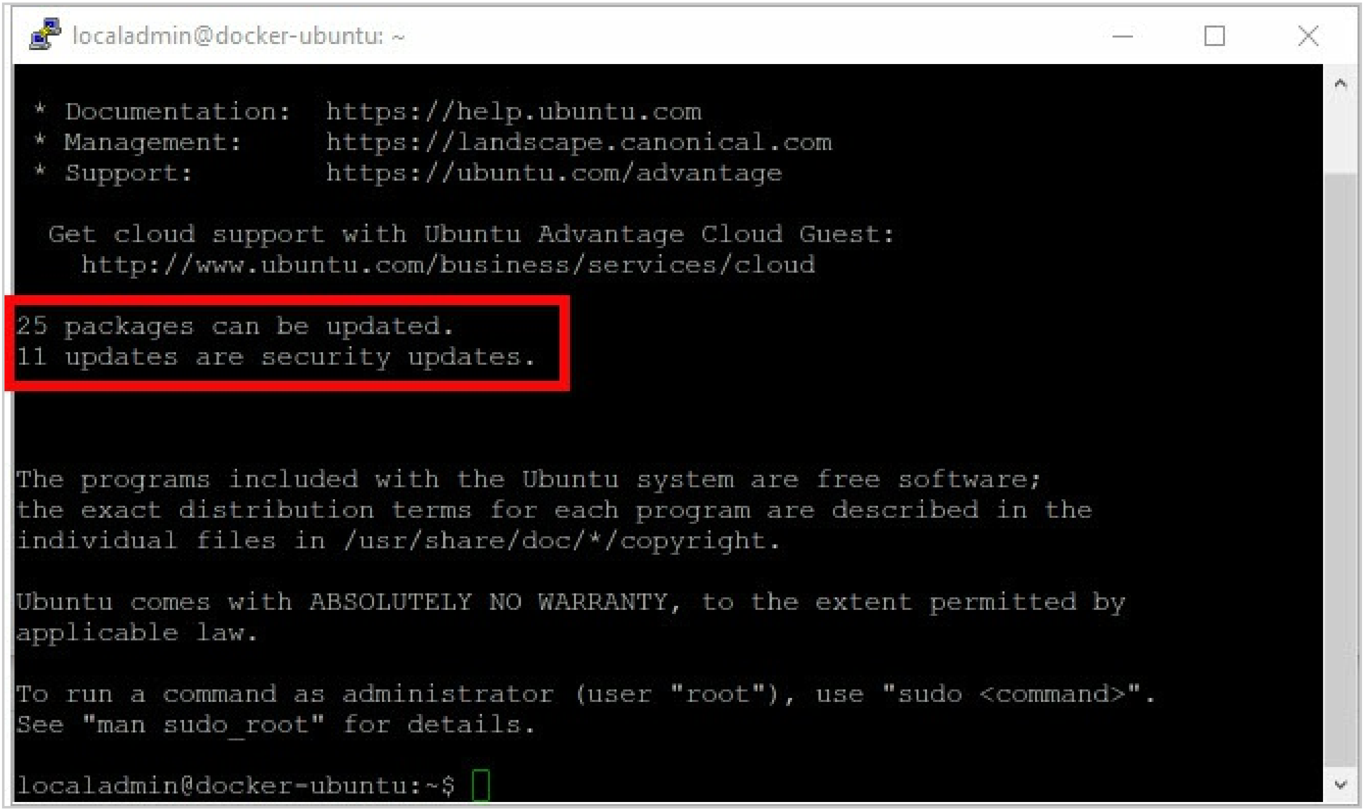
Overview

We have just created our Ubuntu 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 Ubuntu server and login.
2. From the login prompt, you may see a status of available updates. (If not, don't be too alarmed - continue with these steps anyway just to be sure.)



```
 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

25 packages can be updated.
11 updates are security updates.


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

localadmin@docker-ubuntu:~$
```

3. First we need to ensure our list of sources for our system updates are up-to-date. From the

command prompt, type the following:

```
sudo apt-get update
```

4. Now we can install updates. From the command prompt, type the following to automatically install all available updates:

```
sudo apt-get upgrade -y
```

5. Depending on the number and size of available updates, this process may take a few minutes. Now would be a good time to take a break.

Install Docker

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

1. We need to add the GPG key for the official Docker repository to the system because in the next step we want to download the Docker 'installer' directly from Docker and not the default Ubuntu servers to ensure we get the latest version of the engine. From the command prompt, type the following:

```
sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys 5
8118E89F3A912897C070ADBF76221572C52609D
```

Cut & Paste

You can paste this into PuTTY by *right-clicking* the terminal screen.

2. Now, we need to tell Ubuntu *where* the Docker repository is located. From the command prompt, type (or paste) the following:

```
sudo apt-add-repository 'deb https://apt.dockerproject.org/repo ubuntu-xenial m
ain'
```

3. Once again, update the package database with the Docker packages from the newly added repository:

```
sudo apt-get update
```

4. Make sure you are about to install from the Docker repository instead of the default Ubuntu repository:

```
apt-cache policy docker-engine
```

5. You should see output *similar* to the following (notice that `docker-engine` is not installed and the `docker-engine` version number might be different):

```
docker-engine:
Installed: (none)
Candidate: 1.11.1-0~xenial
Version table:
   1.11.1-0~xenial 500
      500 https://apt.dockerproject.org/repo ubuntu-xenial/main amd64 Packages
   1.11.0-0~xenial 500
      500 https://apt.dockerproject.org/repo ubuntu-xenial/main amd64 Packages
```

6. Finally, install Docker:

```
sudo apt-get install -y docker-engine
```

7. Installing the Docker engine may take an additional minute or two.

Additional Configuration

To simplify running and managing Docker, there's some additional configuration that we need to implement. While this section is optional, it is recommended to make managing Docker much easier.

Ensure Docker Engine is Running

1. From the command prompt, type:

```
sudo systemctl status docker
```

2. You should see something similar to the following:

```
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset:
enabled)
   Active: active (running) since Sun 2017-06-04 22:38:16 UTC; 4min 10s ago
     Docs: https://docs.docker.com
 Main PID: 32844 (dockerd)
```

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

Enable Docker Engine at Startup

Let's make sure the Docker engine is configured to run on system startup (and reboot).

1. From the command prompt, type:

```
sudo systemctl enable docker
```

2. You should see something similar to the following:

```
Synchronizing state of docker.service with SysV init with /lib/systemd/systemd-
sysv-install...
Executing /lib/systemd/systemd-sysv-install enable docker
```

Elevate Your Privileges

Be default, running the `docker` command requires root privileges - that is, you have to prefix the command with `sudo` . It can also be run by a user in the **docker** group, which is automatically created during the install of Docker. If you attempt to run the `docker` command without prefixing it with `sudo` or without being in the docker group, you'll get an output like the following:

```
docker: Cannot connect to the Docker daemon. Is the docker daemon running on th
is host?.
See 'docker run --help'.
```

To avoid typing `sudo` whenever you run the `docker` command, add your username to the docker group:

```
sudo usermod -aG docker $(whoami)
```

You will then need to log out and back in for the changes to take effect.

If you need to add another user to the `docker` group (one in which you have not logged in as currently), simply provide that username explicitly in the command:

```
sudo usermod -aG docker <username>
```

You've successfully installed the Docker engine. You have also configured it to run at startup and have added yourself to the **Docker** group so that you have sufficient privileges for running Docker.

Hello World

Overview

Now that we have Docker installed, we are able to deploy images as containers. In this short step of the workshop, we will deploy a couple of small containers as practice.

Hello World

1. Ensure you have logged in to your remote Ubuntu server and are at the prompt.
2. From the command prompt, type the following:

```
docker run hello-world
```

3. You should then see something similar to the following:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

I'll explain what the first couple of lines mean later. The important thing here is to see that around the 7th line, you'll see the message 'Hello from Docker!' followed by a line informing you that the Docker installation 'appears to be working correctly.'

Whalesay

Now let's run another fun container.

1. From the command prompt, type:

```
docker run docker/whalesay cowsay 'Azure Rocks!'
```

2. After downloading the dependent images, you should see an ASCII whale with a speech bubble containing the message 'Azure Rocks!'.

This image is based on the old Unix *cowsay* game. We're basically running *cowsay* in a container and telling the whale to say whatever we provide in single quotes. You can run the command as many times as you'd like and put whatever you'd like the whale to say in single quotes. Go ahead and give it a try. Notice, that after the first time running the container, the image is no longer downloaded. More about this in the next section.

Working With Containers

Overview

We've successfully deployed a couple of containers into our Docker engine. In this section, we'll dig a little deeper into working with and interacting with containers.

Listing Images

As we experienced with running the *Whalesay* container multiple times, the actual image was only downloaded, uncompressed and built once. On all subsequent executions, a new container was simply instantiated based on the image. Docker keeps a local repository of images currently in use; and, those images cannot be deleted until all dependent containers have been deleted.

To view a list of currently downloaded images, type the following in the command prompt:

```
docker images
```

The output should look similar to the following:

```
REPOSITORY          TAG                 IMAGE ID            CREATED
 SIZE
hello-world         latest              48b5124b2768        4 months ago
 1.84kB
docker/whalesay     latest              6b362a9f73eb        2 years ago
 247MB
```

There's a few things that are reported to us here.

First, we see the repository, including the namespace, of the image. The `hello-world` is what we would consider a *library* image. In other words, it's an image that, for lack of a better way to describe it, is 'built-in' to Docker. For `whalesay`, we see that the repository is `docker`. We'll talk more about repositories below.

The second column shows us the current tag of the image. We look at tagging in the next workshop section.

The third column displays a unique id of the image. Just so that you know, we can refer to the image in our commands throughout the exercise by the full name as it's listed under `REPOSITORY` , the image id, or simply use the first 3 characters in the image id (e.g. `hello-world` could also be referenced by `48b5124b2768` or `48b`). The thing is, the minimum is 3 characters, but if you have multiple images that have the same first 3 characters, you may need to use a couple of more until you reach a differentiator.

The created column does not report when you downloaded the image locally. Instead, it reports the date of when the image was created by its owner/designer. This is a great column for DevOps to use when quickly trying to determine when a particular image was created.

The size column reports the total size of the image - a sum of all layers comprised to make the image.

Docker Repositories/Registries

A registry is a service - public or private - from which images can be hosted and pulled by other users. Images are stored in these repositories under namespaces which are, typically, usernames or organizational names.

In the above example, the `whalesay` image is located under the `docker` namespace. This means that the image belongs to and is managed by the Docker organization.

Microsoft's public registry is hosted by Docker. You can visit Microsoft's registry at https://hub.docker.com/u/microsoft/. As you view the available images (which are only available for Windows-based machines), you'll see that each begin with `microsoft/` . If we were to pull an image created and maintained by Microsoft into our local Docker instance, we would see the image prefaced by that namespace.

Inspecting Images

There are a couple of ways we can get some greater details about our images. We can view the underlying metadata of our image; and, we can see the build history of the image.

Image Inspection (Metadata)

To view the metadata of an image, we'll need to *inspect* it. From the command prompt, type the following:

```
docker image inspect docker/whalesay
```

You should see something that begins with the following:

```
[
    {
        "Id": "sha256:6b362a9f73eb8c33b48c95f4fcce1b6637fc25646728cf7fb0679b2da
273c3f4",
        "RepoTags": [
            "docker/whalesay:latest"
        ],
        "RepoDigests": [
            "docker/whalesay@sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3
d31e18aaf55f3258ba93b"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2015-05-25T22:04:23.303454458Z",
        "Container": "5460b2353ce4e2b3e3e81b4a523a61c5adc238ae21d3ec3a577467465
2e6317f",
        "ContainerConfig": {
            "Hostname": "9ec8c01a6a48",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ENV PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Image": "5d5bd9951e26ca0301423625b19764bda914ae39c3f2bfd6f1824bf53
54d10ee",
            "Volumes": null,
            "WorkingDir": "/cowsay",
            "Entrypoint": null,
            "OnBuild": [],
            "Labels": {}
        },
        "DockerVersion": "1.6.0",
        ...
```

Take a moment to examine the metadata. As you look through this information, you will find various attributes that describe the image.

Image History

The last bit of information that the `inspect` command reported was a set of layers:

```
    "Layers": [
        "sha256:1154ba695078d29ea6c4e1adb55c463959cd77509adf09710e2315827d66271
a",
        "sha256:528c8710fd95f61d40b8bb8a549fa8dfa737d9b9c7c7b2ae55f745c972dddac
d",
        "sha256:37ee47034d9b78f10f0c5ce3a25e6b6e58997fcadaf5f896c603a10c5f35fb3
1",
        "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6e
f",
        "sha256:b26122d57afa5c4a2dc8db3f986410805bc8792af3a4fa73cfde5eed0a8e5b6
d",
        "sha256:091abc5148e4d32cecb5522067509d7ffc1e8ac272ff75d2775138639a6c50c
a",
        "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6e
f",
        "sha256:d511ed9e12e17ab4bfc3e80ed7ce86d4aac82769b42f42b753a338ed9b8a566
d",
        "sha256:d061ee1340ecc8d03ca25e6ca7f7502275f558764c1ab46bd1f37854c74c5b3
f",
        "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6e
f"
    ]
```

Other than the layer id's, this doesn't really tell you much. To see the actual build history of the image, type the following command:

```
docker image history docker/whalesay
```

You'll will then see something similar to the following:

```
IMAGE                   CREATED             CREATED BY
         SIZE                    COMMENT
6b362a9f73eb            2 years ago         /bin/sh -c #(nop) ENV PATH=/usr/local/b
in:...   0B
<missing>               2 years ago         /bin/sh -c sh install.sh
         30.4kB
<missing>               2 years ago         /bin/sh -c git reset --hard origin/mast
er       43.3kB
<missing>               2 years ago         /bin/sh -c #(nop) WORKDIR /cowsay
         0B
<missing>               2 years ago         /bin/sh -c git clone https://github.com
/mo...   89.9kB
<missing>               2 years ago         /bin/sh -c apt-get -y update && apt-get
 in...   58.6MB
<missing>               2 years ago         /bin/sh -c #(nop) CMD ["/bin/bash"]
         0B
<missing>               2 years ago         /bin/sh -c sed -i 's/^#\s*\(deb.*univer
se\...   1.9kB
<missing>               2 years ago         /bin/sh -c echo '#!/bin/sh' > /usr/sbin
/po...   195kB
<missing>               2 years ago         /bin/sh -c #(nop) ADD file:f4d7b4b3402b
5c5...   188MB
```

Keep in mind that these are *layers*. So the way to read this is from bottom-up, meaning that the last row is actually the *first* layer. Then, as you go up the list, additional layers are applied. The bottom layer is usually a base image, such as an OS and, because it's the OS, is usually larger in size. Then, actions or commands are applied to that image that make up the additional layers. Some of the last actions to be applied to this image were running an install script and setting some environment variables.

Let's take a look at another example. Download another image to your local Docker engine by typing the following command:

```
docker image pull a11smiles/softcover
```

This is an image stored under my personal namespace in the public Docker registry. Softcover is a ruby-based application used to build digital books for various platforms. Running this command will take a few minutes, but, as you will observe, there's quite a few layers to this image.

Once this image download and reconstruction has completed, check out the history of the image:

```
docker image history a11smiles/softcover
```

You will see all of the commands, which include installing multiple dependency libraries for Softcover (e.g. `apt-get install ...`), issued to build the image. Some of these dependencies are quite large, ranging from 815kB to almost 4GB. Image the time it would require to download all of these requirements manually. This doesn't include the time it would take to properly and consistently configure them for each developer and/or environment. With the Softcover container, in this case, simply download the image and run the container with supplying your book's source to create a digital publication. This can be easily wired up in an automated build process.

Listing Containers

Not only can we list our local images, but we can also list our local, instantiated containers.

Running Containers

Type in the command line:

```
docker ps
```

You should see some like the following:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
 STATUS              PORTS               NAMES
```

This tells us that no containers are currently running.

Non-running Containers

We can see previously ran containers by typing in the following:

```
docker ps -a
```

This will render a report similar to the following:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
 STATUS                     PORTS               NAMES
a883ff18a967        docker/whalesay     "cowsay Hola!"      26 hours ago
 Exited (0) 26 hours ago                        sad_goldstine
baa0591c4392        hello-world         "/hello"            26 hours ago
 Exited (0) 26 hours ago                        jovial_raman
```

You may see more depending on how many times you instantiated the `whalesay` image. The important thing to see from this report is the *Status* of containers. All containers, in this report, have exited and are no longer running.

One thing to note is that even though the containers have finished executing, they have **not** been deleted. This allows you to enter into a stopped container. One reason this may be useful is if an application has thrown an exception and quit, you may want to enter into the container to check out some log files or something of that nature to debug the application. Once you're done with the container, you must delete it. We will see how to do this in the next page of the workshop.

Running a Container

We've already instantiated a couple of images. We do this by the following command:

```
docker run <namespace/image>
```

There are two primary types of processes - short-running and long-running.

Short-running Processes

Short-running process are just like the images we've already ran - `hello-world` and `whalesay` . Another example is the `softcover` image. A short-running process typically runs a single command, performing a single function, then exits. This function could be displaying a message, sending an email, or building an application or other resource (e.g. a book, help documentation, etc).

Every time we we run a short-running process, using the above command another container is instantiated. This is why a new container was created ever time you ran `whalesay` .

Long-running Processes

Long-running processes are typically service-based applications (e.g. web servers, service buses, queues, etc.). We can also run an OS as a container. Try typing in the following:

```
docker run -it ubuntu /bin/bash
```

This command will will download an `ubuntu` image and run the `bash` shell in *interactive* mode (`-i`) by opening up a *terminal* (`-t`). When this runs, you will be automatically placed inside of the running Ubuntu container (notice the change in the command prompt (e.g. something similar to `root@<container id>:/#`)). If you are familiar with Ubuntu, feel free to take a look around. To exit the container, simply type in `exit` . This will return you to your host machine.

Let's now run Ubuntu in *detached* mode. Instantiating a container in interactive mode assumes we are going to interact with the running container. However, typical long-running processes (like web servers) don't require command line interaction. We'll run a web server later in this workshop. However, for the moment, let's mimic a long-running process by running Ubuntu in an infinite sleep mode. Execute the following command:

```
docker run -d ubuntu sleep inf
```

Again, this will run an Ubuntu container in *detached* mode (`-d`) and set it to sleep infinitely.

Let's see the container running in the background by typing in:

```
docker ps
```

You should see something like:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
 STATUS              PORTS               NAMES
094012b145c8        ubuntu              "sleep inf"         2 seconds ago
 Up 2 seconds                            vigorous_yalow
```

You'll see under the *Status* column, that it's reporting an up-time for the container. We don't typically run commands against long-running processes, but just for the practice, issue the following command (substitute the container id with your id):

```
docker exec 094 ls
```

This will run a `list` command against the current directory *inside* of the container and return a list of subfolders.

While the container is running, let's look at one more interesting thing. At the command prompt, type the following:

```
ps aux
```

This will list all of the currently running processes. As you scroll and look at the background processes, you should see a line similar to the following. The process id, memory consumption, etc. may be different but try to find the last column.

```
root     55515  0.0  0.0   4380   664 ?        Ss   01:39   0:00 sleep inf
```

What you see here is that container processes are exposed to the host kernel and have access to host system resources.

Now let's stop the container. Technically, we could kill the process using traditional Linux (UNIX) commands, but this would leave our Docker engine in an unclean state and would require us to do some additional clean up. Let's stop the container using Docker commands.

```
docker stop 094
```

(NOTE: Again, `094` is the first 3 characters of my container's id. You'll need to use the id assigned to your container.)

Running `docker ps` again should show you that no more containers are currently running.

Re-running a Stopped Container

There will be times when you may need to re-run a stopped container. Find one of the stopped 'whalesay' containers by typing the following:

```
docker ps -a
```

My output looks like the following:

```
CONTAINER ID        IMAGE               COMMAND             CREATED
 STATUS                        PORTS               NAMES
094012b145c8        ubuntu              "sleep inf"         27 minutes ago
 Exited (137) 4 minutes ago                        vigorous_yalow
a883ff18a967        docker/whalesay     "cowsay Hola!"      27 hours ago
 Exited (0) 38 minutes ago                         sad_goldstine
baa0591c4392        hello-world         "/hello"            27 hours ago
 Exited (0) 27 hours ago                           jovial_raman
```

By typing the following command, I can re-run my `whalesay` container *without* instantiating a new container and without having to supply any parameters. Basically, I can re-run the container as it was originally ran.

```
docker start -a a88
```

(NOTE: Again, `a88` is the first 3 characters of the stopped `whalesay` container's id.)

With this command, we are restarting a stopped container and attaching (`-a`) to the container's output (STDOUT/STDERR) to view any messages. In our case, the message is a simple whale with a speech bubble.

Wow, congratulations! You just successfully completed what may seem like a crash course in managing containers. However, there's really not much more to it than this. As with anything, the more you play with Docker, the more familiar and confident you become with it.

Image and Container Administration

Overview

You've just learned quite a bit in working with containers. As a matter of fact, the majority of container management has already been covered. We're now going to bring our knowledge of container and image management around full-circle. This will complete the administration portion of the workshop.

Tagging Images

There will be instances, like DevOps for instance, where we will need to *tag* our images. Tags allow us to apply labels to our images. This is useful for tracking changes, such as in versioning, to our images.

Let's create a simple derivative of the Ubuntu image we downloaded previously.

Instantiate a new Ubuntu container by typing the following:

```
docker run -it ubuntu /bin/bash
```

This will place you at the command prompt *inside* of the running container. Now, let's interact with the OS by typing the following commands.

```
cd
mkdir test
cd test
echo "This is some sample text." > test.txt
exit
```

We've just created a new directory with a test text file in the user's home directory. If I view my available containers (`docker ps -a`), I'll find the id of the container I just exited from (look under the *Status* column for the container that was just exited). In my case (see the following output), the container's id is `335abd61d52d` .

```
CONTAINER ID        IMAGE               COMMAND             CREATED
 STATUS                     PORTS               NAMES
335abd61d52d        ubuntu              "/bin/bash"         3 minutes ago
 Exited (0) About a minute ago                  blissful_wing
094012b145c8        ubuntu              "sleep inf"         About an hour ago
 Exited (137) 27 minutes ago                    vigorous_yalow
a883ff18a967        docker/whalesay     "cowsay Hola!"      27 hours ago
 Exited (0) 17 minutes ago                      sad_goldstine
baa0591c4392        hello-world         "/hello"            27 hours ago
 Exited (0) 27 hours ago                        jovial_raman
```

Let's restart the container and check to make sure our text file is still there (just to confirm). Again, replace the id below with your container's id.

```
docker start -i 335
```

At the prompt type in:

```
ls ~/test
```

This should list a file named `test.txt` . Now, type `exit` to exit out of the container.

We now have a customized container based on our Ubuntu image. Let's create our own image with it's tag. We're also going to add a message and an author to the image's metadata. Once again, replace the `335` below with the id of your stopped container.

```
docker commit -m "added test.txt" -a "Some User" 335 mynamespace/testtext:v1
```

With this command, again, I'm added a message (`-m`) to describe the image and an author (`-a`) to inform of the author. The `335` is the first 3 characters of my stopped, modified container. Finally, I've supplied a namespace (`mynamespace/`), an image name (`testtext`), and a tag (`v1`).

The namespace is optional, but a good practice to differentiate between images that might have the same name. For example, if you and another developer are working on two separate images and you have them both locally, it's easier to keep track of who's image belongs to who.

Now execute the following command:

```
docker images
```

You should see something similar to the following:

```
REPOSITORY              TAG                     IMAGE ID            CREATED
    SIZE
mynamespace/testtext    v1                      556c25bff4b1        5 minutes ago
    118MB
ubuntu                  latest                  7b9b13f7b9c0        3 days ago
    118MB
a11smiles/softcover     latest                  306f23683872        3 months ago
    5.74GB
hello-world             latest                  48b5124b2768        4 months ago
    1.84kB
docker/whalesay         latest                  6b362a9f73eb        2 years ago
    247MB
```

Notice that you now have your custom image with its tag. Also, because our text file isn't very large, our image has, virtually, the same size as that of the `ubuntu` image (118MB).

We can then instantiate a container based on our image by running the following:

```
docker run -it mynamespace/testtext:v1 /bin/bash
```

(NOTE: In all the previous times we've run this command, we've never had to specify a tag because the `latest` tag is implied. In our case, the tag we used is `v1` so we have to specify it.)

This will place us, once again, inside the container. Run the following command in the container:

```
cat ~/test/test.txt
```

This will show the contents of the file we added earlier.

Now we can exit out of the container by simply typing in `exit` .

In the host machine, typing in `docker ps -a` shows us that our custom image instantiated a container which just exited successfully.

```
CONTAINER ID        IMAGE                          COMMAND             CREATED
       STATUS                        PORTS               NAMES
a215acbb7981        mynamespace/testtext:v1   "/bin/bash"         3 minutes ago
       Exited (0) 13 seconds ago                         inspiring_spence
335abd61d52d        ubuntu                    "/bin/bash"         26 minutes ag
o      Exited (0) 16 minutes ago                         blissful_wing
094012b145c8        ubuntu                    "sleep inf"         About an hour
 ago   Exited (137) About an hour ago                    vigorous_yalow
a883ff18a967        docker/whalesay           "cowsay Hola!"      28 hours ago
       Exited (0) 39 minutes ago                         sad_goldstine
baa0591c4392        hello-world               "/hello"            28 hours ago
       Exited (0) 28 hours ago                           jovial_raman
```

Finally, if we inspect our custom image (`docker image inspect mynamespace/testtext:v1`), we will see the comment and author attributes displaying "added test.txt" and "Some User", respectively. And, the top layer of our history (`docker image history mynamespace/testtext:v1`) shows us entering into the `bash` shell.

Deleting Containers

We can clean up disk space by removing unused containers and images. However, we cannot remove any images that currently have dependent containers - even containers that have stopped. Therefore, we must delete dependent containers first.

For our example, let's suppose that we no longer need the `ubuntu` container anymore because we've customized it (e.g. added our own text file). We can delete the `ubuntu` container by typing the following:

```
docker rm 335
```

Let's also delete our `hello-world` container:

```
docker rm baa
```

Remember, for the previous two commands, substitute your respective container ids.

Deleting Images

Deleting images are just as easy. First, let's refresh ourselves on our locally installed images. Running `docker images` produces the following output:

```
REPOSITORY              TAG                     IMAGE ID            CREATED
    SIZE
mynamespace/testtext    v1                      556c25bff4b1        19 minutes ago
    118MB
ubuntu                  latest                  7b9b13f7b9c0        3 days ago
    118MB
a11smiles/softcover     latest                  306f23683872        3 months ago
    5.74GB
hello-world             latest                  48b5124b2768        4 months ago
    1.84kB
docker/whalesay         latest                  6b362a9f73eb        2 years ago
    247MB
```

Since images, combined with their namespaces and tags, are unique on the local Docker engine, we can delete images by using the full namespace reference (including the tag) or by using the image id. Let's practice deleting images.

First, let's delete the `hello-world` image:

```
docker rmi hello-world
```

As a reminder, the `latest` tag is implied. If we were to delete our custom image, we would be required to supply the tag because it differs from `latest` .

Running the `docker rmi` command will remove any links between the image and shared layers. If the layer is no longer required by any other image, the layer is also deleted.

Now, let's attempt to delete the `ubuntu` image:

```
docker rmi ubuntu
```

Running this command produces and error - namely, that the image cannot be deleted because there's still a container that depends on it. Running `docker ps -a` shows that this, indeed, is the case (the second container listed below):

```
CONTAINER ID        IMAGE                     COMMAND             CREATED
       STATUS                        PORTS               NAMES
a215acbb7981        mynamespace/testtext:v1   "/bin/bash"         20 minutes ag
o      Exited (0) 17 minutes ago                         inspiring_spence
094012b145c8        ubuntu                    "sleep inf"         About an hour
 ago   Exited (137) About an hour ago                    vigorous_yalow
a883ff18a967        docker/whalesay           "cowsay Hola!"      28 hours ago
       Exited (0) About an hour ago                      sad_goldstine
```

One of the many reasons for this, is to protect against accidental deletion of our containers and images. However, if are sure we want to delete the image and all its containers, we can *force* a deletion:

```
docker rmi -f ubuntu
```

Besides forcing a delete of the image, notice how the output is different from the previous deletion of the `hello-world` image. In this last case, only the reference, or link, was removed from the image. The underlying layers weren't *deleted*. Why? Because the custom image that we created earlier still depends on the underlying Ubuntu OS layer(s). This is one way Docker helps to conserve disk space - shared and reuse of dependencies. Deleting our custom image (and containers) would perform an actual delete of the Ubuntu OS layer(s).

Exposing Services In Azure

Overview

The final part of this workshop is to practice exposing a container service outside of Azure. We're going to create a simple web server and access it from our local machine.

NGINX

We are going to deploy a container hosting NGINX (pronounced "engine X"), a simple, but powerful web server. NGINX is typically used in containerized deployments because it supports autoscaling, service discovery and other capabilities often leveraged in microservice architecture.

Run NGINX by typing the following command:

```
docker run -d -p 8080:80 nginx
```

This will download and run NGINX in the background. As stated earlier in this workshop, we often run services in *detached* mode (`-d`). As new parameter that you see here is mapping, or publishing (`-p`), ports - very similar to a NAT, if you are familiar with the concept. There are two ports specified here separated by a colon. The first number is the host's port while the second number is the container's port. So, in essence, we are mapping the host's port 8080 to the container's port 80. If our container runs multiple services or a service requiring multiple ports, we can also specify a port range. We could have used the default HTTP port 80 for the host, but for the workshop I chose port 8080 for differentiation between the two environments in pursuit of clarity.

Let's make sure that NGINX is running successfully.

```
curl http://localhost:8080
```

Running the previous command, should display some html source code.

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Network Security Group (NSG)

Now that our web server is running, let's make it available outside of Azure.

When we created our Ubuntu virtual machine, we accepted the defaults, including the default settings for our NSG. The default settings only allowed SSH (port 22) access. We need to add a rule to our NSG to allow HTTP traffic over port 8080.

1. If you are not still there, go back to the Azure portal and navigate to the settings of your Ubuntu virtual machine.
2. In the left menu, click on **Network interfaces** .
3. This will open the *Network Interfaces* blade for your Ubuntu 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-ubuntu-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:

    - Name: **allow-http**
    - Priority: **1010**
    - Source: **Any**
    - Service: **Custom**
    - Protocol: **Any**
    - Port range: **8080**
    - Action: **Allow**
9. Click **OK**.

This should only take a couple of seconds. Once you see the rule added, open a new browser and navigate to the IP address of your Ubuntu virtual machine, including the port number. The IP address used in this workshop's screen shots is **40.121.213.77** (your IP address will be different). Using the aforementioned IP address, I would direct my browser to **http://40.121.213.77:8080**. Doing so, you should see the NGINX landing page.