

Docker Automation with Dockerfiles (Linux)

Friday, August 4, 2017 4:52 PM

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

This workshop will show you how to automate the building and configuring of a Linux-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 Linux-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 Linux-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: 45 minutes

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.

Introduction

Overview

This workshop contains two routes - one for constructing a Dockerfile in Ubuntu and another for CentOS. The steps are primarily the same, but the separate sections were provided to minimize confusion.

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.

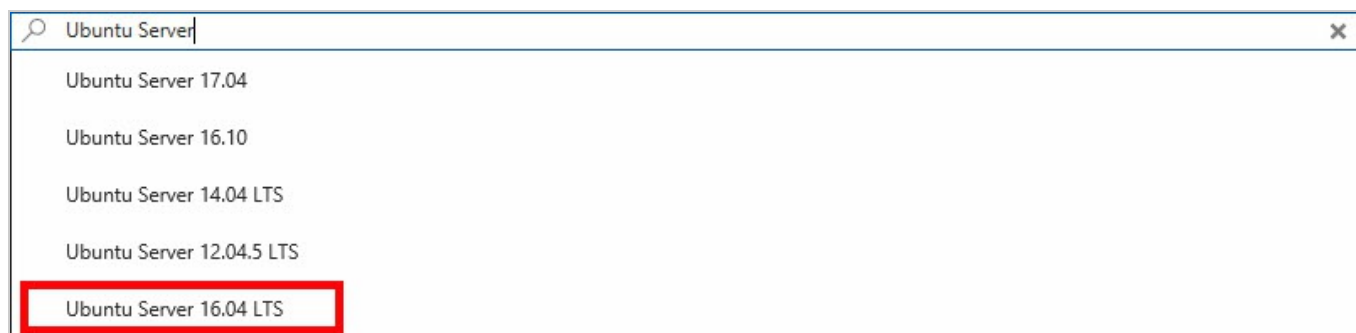
1. If you are not there already, go ahead and click on the **Resource Groups**  in the Azure Portal to open the Resource Groups blade.
2. At the top of the Resource Groups blade, click on **Add** . This will open a panel that asks for some basic configuration settings.
3. Complete the configuration settings with the following:
 - Resource group name: **azworkshops_dockerfile_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_dockerfile_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**.



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: **dockerfile-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_dockerfile_ubuntu_demo**
 - Location: *<choose a location>*
 2. Size
 - **DS1_V2**
 3. Settings
 - Use managed disks: **No**
 - Storage account: (click on it & **Create New**)

- Name: **dfubuntudata**<random number> (ex. *dfubuntudata123456*) (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_ubuntu_demo-vnet*)
 - Subnet: <accept default> (e.g. *default (172.16.1.0/24)*)
 - Public IP address: <accept default> (e.g. *(new) dockerfile-ubuntu-ip*)
 - Network security group (firewall): <accept default> (e.g. *(new) dockerfile-ubuntu-nsg*)
 - Extensions: **No extensions**
 - Availability set: **None**
 - Boot diagnostics: **Enabled**
 - Guest OS diagnostics: **Disabled**
 - Diagnostics storage account: (click on it & **Create New**)
 - Name: **dfubuntudiags**<random number> (ex. *dfubuntudiags123456*)
 - 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.

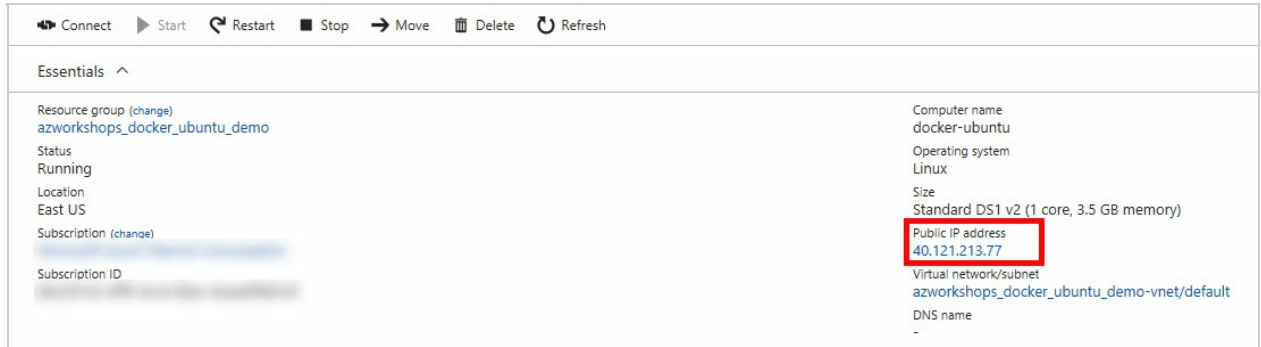
Connect to the Virtual Machine

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.

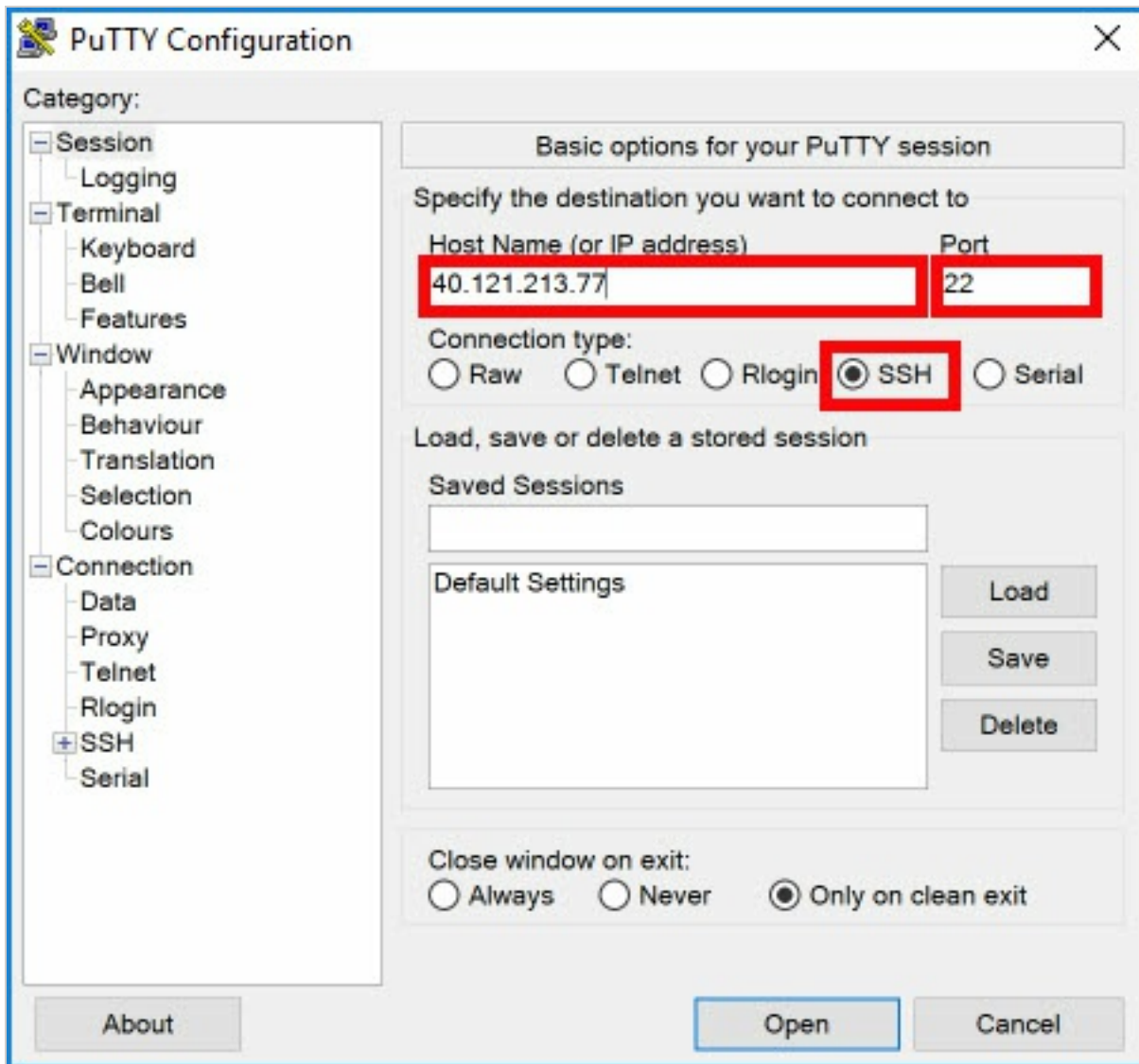


The screenshot shows the 'Essentials' section of an Azure VM blade. On the left, there are links for 'Resource group (change)', 'azworkshops_docker_ubuntu_demo', 'Status', 'Running', 'Location', 'East US', 'Subscription (change)', and 'Subscription ID'. On the right, the VM's configuration is listed: 'Computer name: docker-ubuntu', 'Operating system: Linux', 'Size: Standard DS1 v2 (1 core, 3.5 GB memory)', 'Public IP address: 40.121.213.77' (highlighted with a red box), 'Virtual network/subnet: azworkshops_docker_ubuntu_demo-vnet/default', and 'DNS name: -'. At the top, there is a toolbar with icons for 'Connect', 'Start', 'Restart', 'Stop', 'Move', 'Delete', and 'Refresh'.

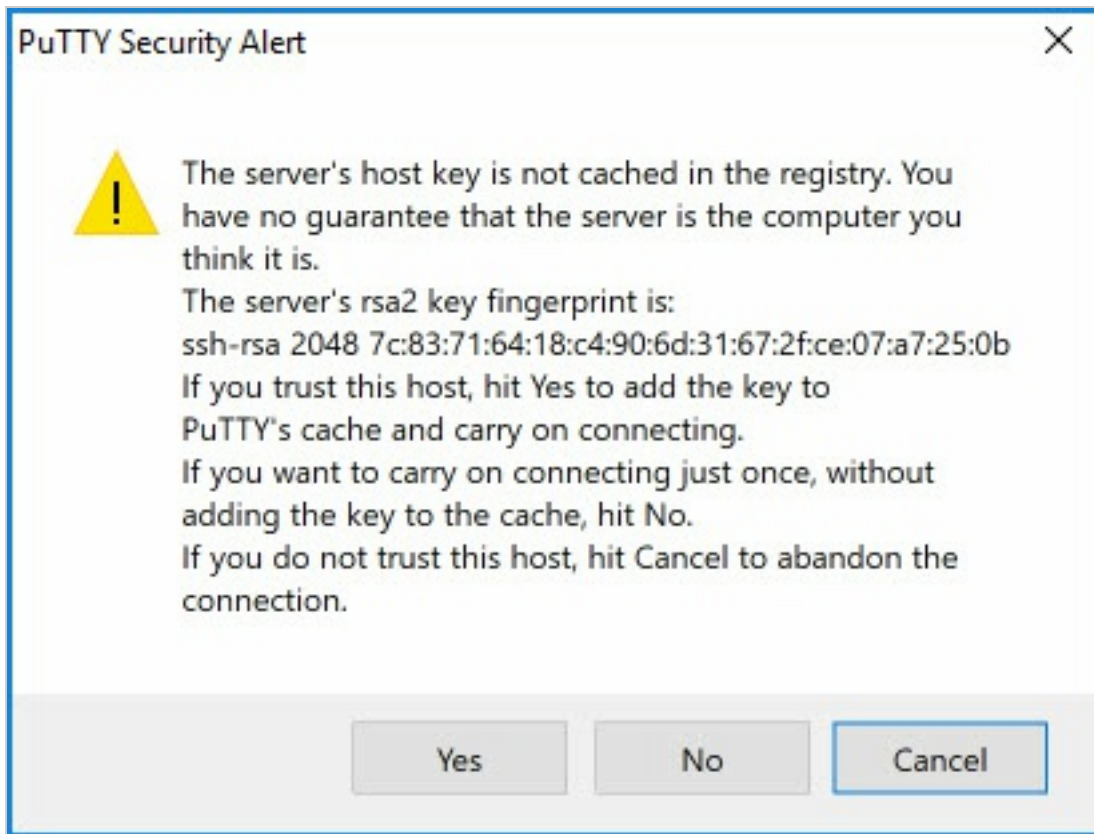
3. Copy the IP address.

Connect with SSH

1. Open **PuTTY**.
2. In the configuration window:
 - o Hostname: **<IP address from previous step>**
 - o Port: **22**
 - o Connection type: **SSH**



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



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@dockerfile-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 Updates

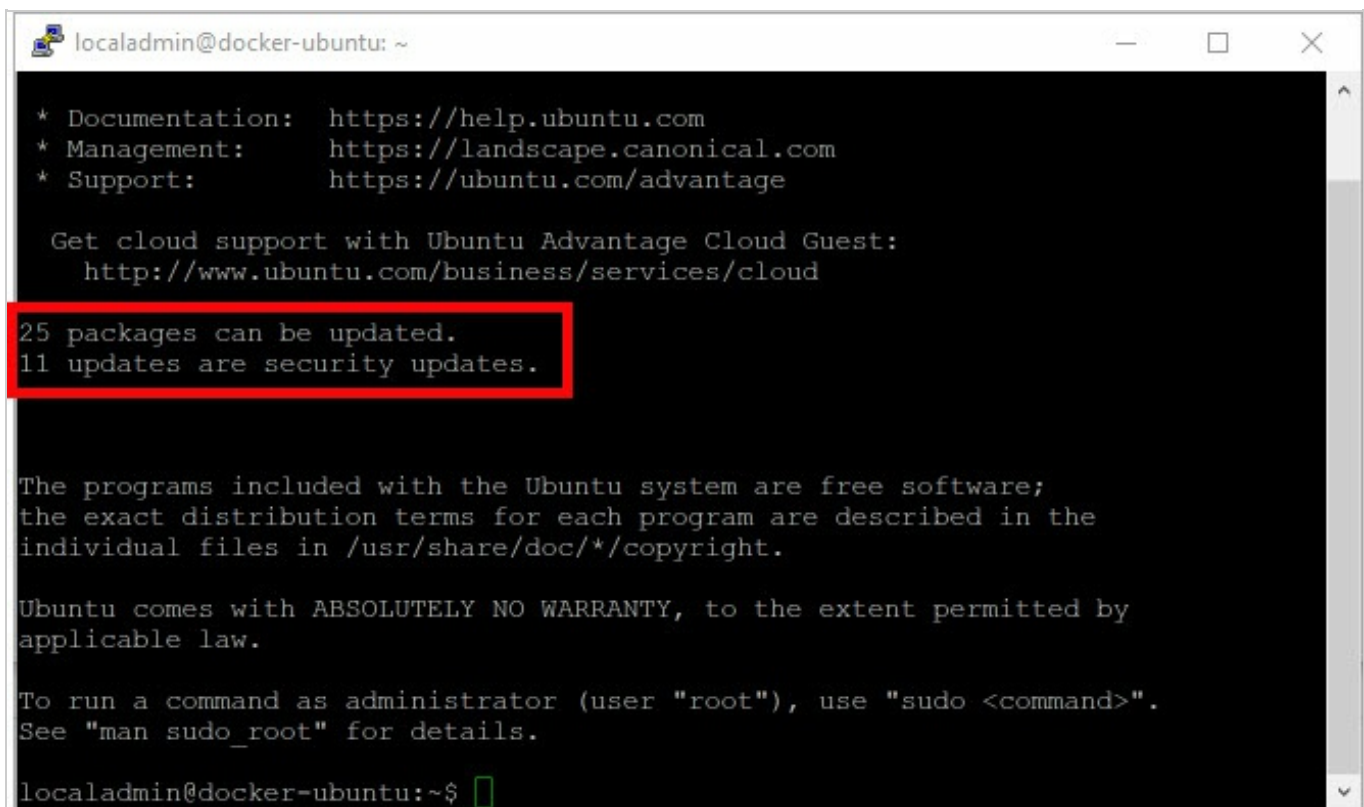
Overview

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.)

A terminal window titled 'localadmin@docker-ubuntu: ~' with standard window controls. The terminal output shows Ubuntu system information, including documentation, management, and support links. A red box highlights the update status: '25 packages can be updated.' and '11 updates are security updates.' Below this, there is a disclaimer about Ubuntu being free software and a note about running commands as administrator using 'sudo'. The prompt 'localadmin@docker-ubuntu:~\$' is visible at the bottom.

```
localadmin@docker-ubuntu: ~
* 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

Overview

In this step, we are going to install Docker. This is one of the steps that will actually *not* be performed inside of the container image. But, of course, we need Docker on the host in order to run containers.

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 58118E89F3A912897C070ADBF76221572C52609D
```

Copy & 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 main'
```

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

By 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.

Install Node.js

Overview

Now that we have the host machine configured for Docker, we're going to begin building our web server. Keep in mind that this virtual machine is serving two purposes: 1) our Docker host; and, 2) our temporary web server until our image is built.

Install Node.js

Node.js is a JavaScript execution engine that allows us to write server-side JavaScript and utilize it like any other executable file. With Node.js, we will host a very simple web server.

Download and Install Node.js Runtime

The Node.js runtime is what is responsible for hosting and executing our JavaScript.

From the terminal prompt, type:

```
sudo apt-get install -y nodejs
```

Download and Install NPM

NPM is an abbreviation for *Node Package Manager*. NPM allows us to install Node.js dependencies for our applications.

At the prompt, type:

```
sudo apt-get install -y npm
```

Create Symbolic Link

In Linux, a symbolic link is nothing more than a 'virtual pointer' to a file. It's a way to give a file an alias. For us, we going to do two things with a symbolic link. First, we're going to get `nodejs` an alias. Second, we're going to make the executable globally available so that it can be called from any folder.

At the prompt, enter:

```
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

This will create a pointer called `node` that points to `nodejs`. We can then execute Node.js via either command. However, `node` is the common name to use.

Verify Installation

Let's make sure our installation was successful.

At the prompt, type:

```
node -v  
npm -v
```

You should receive version numbers for both commands.

Add Build Dependencies

Some Node.js libraries have outside dependencies that aren't necessarily installed in Ubuntu by default. Let's install those now.

At the terminal prompt, type:

```
sudo apt-get install build-essential
```

We've now successfully completed the installation of Node.js and the Node Package Manager.

Download Sample Website

Overview

We've downloaded most of our dependencies. In this step we will download a demo web site from GitHub, a remote source code repository.

Clone the Website

'Cloning' is the method in which you download a remote repository to your local machine. We are going to clone the demo web site to the default web hosting folder on our system.

From the command prompt, type the following:

```
sudo git clone https://github.com/AzureWorkshops/samples-simple-nodejs-website.  
git /var/www
```

You should see something similar to the following:

```
Cloning into '/var/www'...  
remote: Counting objects: 9, done.  
remote: Compressing objects: 100% (7/7), done.  
remote: Total 9 (delta 0), reused 6 (delta 0), pack-reused 0  
Unpacking objects: 100% (9/9), done.  
Checking connectivity... done.
```

The website source code has now been downloaded.

Download Dependencies

Overview

Even though we've installed *system* dependencies, our website has a couple of library dependencies that we need to install. We'll also ensure that the website is functioning correctly, before we proceed to build our Dockerfile.

Download Website Dependencies

We'll need to enter into the website's main folder to install the dependencies.

Type the following into the terminal:

```
cd /var/www  
sudo npm i
```

You should see a number of progress bars displayed as dependencies are downloaded followed by what looks like a folder hierarchy.

Test Our Website

The first thing we need to do is run our web hosting server which is handled by Node.js. If we ran this normally from the command-line, it would block all other input. We would then have to open a second terminal to test our website. Instead, we're going to run our web server in the background as a detached process.

At the terminal prompt, type:

```
sudo node index.js &
```

NOTE: The ampersand at the end of the line instructs the system to run the preceding command in the background. When we construct our Dockerfile, we will omit the ampersand so that the process runs in the foreground and creates a long-running process that keeps our container 'alive'.

When you run this command, a number is outputted to the screen. This number refers to a *process id* (pid). Keep this number handy as we'll use it below to 'kill' the background process.

Additionally, you should see a message stating that our 'server is listening on 8080'.

Test that the web server is returning our site by typing the following:

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

If all is successful, you should receive some HTML source code back.

Stop the Background Process

We don't need the website running any longer now that we know it works. Additionally, if we kept it running, it's use of the port 8080 could create potential conflicts if another service (e.g. container) is needing that port.

Referring to the process id (pid) above, execute the following command with your id:

```
sudo kill <pid>
```

If successful, you should see the following confirmation message:

```
[1]+  Done                  sudo node index.js
```

Building our web server has been successful. We are now ready to move forward and replicate our work in building out an image.

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 Ubuntu
2. Update the package references
3. Install the latest packages
4. Install and configure Docker
5. Install Node.js
6. Install Node Package Manager (NPM)
7. Download (clone) the sample website
8. Download website dependencies
9. Run the web server

As a reminder, since we are constructing an image, we can **ignore** step 4. We won't need Docker installed inside of the image.

Create the Dockerfile

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

1. Return to your home folder by typing at the terminal prompt, `cd ~`.
2. Create a Dockerfile using the `nano` text editor by typing the following: `nano Dockerfile`. Nano is reminiscent of the old DOS editor. Of course, you can use `vim` instead if you are comfortable in doing so.
NOTE 'Dockerfile' is case-sensitive.
3. Enter the following without the line numbers. The line numbers are provided for reference below.

```
1 FROM ubuntu:latest
2 MAINTAINER Your Name <you@yourcompany.com>
3
4 RUN apt-get update
5 RUN apt-get upgrade
6 RUN apt-get install -y nodejs
7 RUN apt-get install -y npm
8 RUN ln -s /usr/bin/nodejs /usr/bin/node
9 RUN apt-get install -y git
10
11 RUN git clone https://github.com/AzureWorkshops/samples-simple-nodejs-website.git /var/www
12 WORKDIR /var/www
13 RUN npm i
14
15 EXPOSE 8080
16
17 CMD node /var/www/index.js
```

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

5. To exit, **Ctrl+X**

Explanation

First, if you remember from the previous steps, we prepended each command with `sudo` to allow the command to be executed with elevated privileges. By default, all Docker images execute under the identity of the built-in superuser account `root`. Therefore, we can omit the `sudo`.

Line 1: Specifies the base image, including the tag, with which we're starting. In our case, we are using the minimal Ubuntu OS as the base image.

Line 2: Specifies the owner of the image with their email address.

Lines 4-8: The commands we executed earlier in this workshop that update the system and install Node.js.

Line 9: We were not required to install `git` in our virtual machine. However, because the base image of Ubuntu doesn't include `git` by default, we need to install it manually.

Line 11: Downloads (clones) the sample website into the `/var/www` local folder.

Line 12: `WORKDIR` is how you change the current directory (compared to `cd`) in a Dockerfile. We are changing to the website home directory.

Line 13: Install the website's dependencies.

Line 15: Our website server is programmed to use port 8080. 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.

Line 17: This starts our web server. We could have used the `RUN` directive, but the `CMD` directive is designed to execute our long-running process. While, technically, we could have multiple `CMD` lines in the Dockerfile, the Docker build will ignore all `CMD` lines except the last one.

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 command prompt, type `cd ~` to ensure you are in your home folder, then type the following:

```
docker build -t test/simpleweb .
```

This will build an image using `test/simpleweb` as the repository name. The period at the end specifies the path where Docker can find the Dockerfile.

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 486MB	latest	0bfaff1a6a2a	41 seconds ago
ubuntu 118MB	latest	7b9b13f7b9c0	4 days ago

Our image has been built using the specified repository name. You'll also notice that the `ubuntu` image has been downloaded. This is because the build process required Ubuntu in order to build our image. Now that our image has been built, you could delete the `ubuntu` image if you wanted to. Finally, when looking at the image sizes, you'll see that our image is 4 times larger due installation of Node.js, Git, and the other dependencies. In the end, however, 500MB is still not that large.

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

We've started 3 separated instances of our web server. We've bound the web server's internal port 8080 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	"/bin/sh -c 'node ...'"	3 seconds ago
Up 2 seconds	0.0.0.0:8082->8080/tcp	web_8082	
323a65fa5143	test/simpleweb	"/bin/sh -c 'node ...'"	11 seconds ago
Up 10 seconds	0.0.0.0:8081->8080/tcp	web_8081	
7d4fee5c8f89	test/simpleweb	"/bin/sh -c 'node ...'"	About a minute ago
Up 59 seconds	0.0.0.0:8080->8080/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 are left with successfully creating three container instances running our custom image.

Expose Site 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.

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 our three ports (8080-8082) so that we can access all three containers.

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 - **dockerfile-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:
 - o Name: **allow-http**
 - o Priority: **1010**
 - o Source: **Any**
 - o Service: **Custom**
 - o Protocol: **Any**

- Port range: **8080-8082**
- 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 **52.170.85.112** (your IP address will be different). Using the aforementioned IP address, I would direct my browser to **http://52.170.85.112:8080**. I would also test the other 2 port numbers (e.g. 8081, 8082). You should see the 'Hello World' page at all three URL/port combinations.




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_centos_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 CentOS server.

1. If you are not there already, go ahead and navigate to the **azworkshops_dockerfile_centos_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 CentOS server. Therefore, in the *Search Everything* box, type in **CentOS-based**. This will display a couple of different versions. Since we want to deploy the latest *stable* version of CentOS, from the displayed options, choose **CentOS-based 7.3**.



4. Choose the image published by "Rogue Wave Software".

CentOS-based 7.3		
NAME	PUBLISHER	CATEGORY
 CentOS-based 7.3	Rogue Wave Software (formerly Ope...	Compute
 CentOS-based 7.2	Rogue Wave Software (formerly Ope...	Compute
 Hardened Jenkins on CentOS 7.3	Cognosys Inc.	Compute
 Hardened Wordpress on CentOS 7.3	Cognosys Inc.	Compute
 Hardened Abantecart on CentOS 7.3	Cognosys Inc.	Compute
 Hardened MySQL 5.6 on CentOS 7.3	Cognosys Inc.	Compute
 Hardened Simple Machines on CentOS 7.3	Cognosys Inc.	Compute
 Hardened MYSQL 5.7 on Centos 7.3	Cognosys Inc.	Compute
 Hardened Limesurvey on Centos 7.3	Cognosys Inc.	Compute
 MarkLogic 9.0-1.1 Developer	MarkLogic	Compute
 MarkLogic 9.0-1.1 BYOL	MarkLogic	Compute

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

- Basics

- Name: **dockerfile-centos**
- 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_dockerfile_centos_demo**
- Location: *<choose a location>*

- Size

- **DS1_V2**

- Settings

- Use managed disks: **No**
- Storage account: (click on it & **Create New**)
 - Name: **dfcentosdata**<random number> (ex. *dfcentosdata123456*) (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_centos_demo-vnet*)
- Subnet: <accept default> (e.g. *default (172.16.1.0/24)*)
- Public IP address: <accept default> (e.g. *(new) dockerfile-centos-ip*)
- Network security group (firewall): <accept default> (e.g. *(new) dockerfile-centos-nsg*)
- Extensions: **No extensions**
- Availability set: **None**
- Boot diagnostics: **Enabled**
- Guest OS diagnostics: **Disabled**
- Diagnostics storage account: (click on it & **Create New**)
 - Name: **dfcentosdiags**<random number> (ex. *dfcentosdiags123456*)
 - 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.

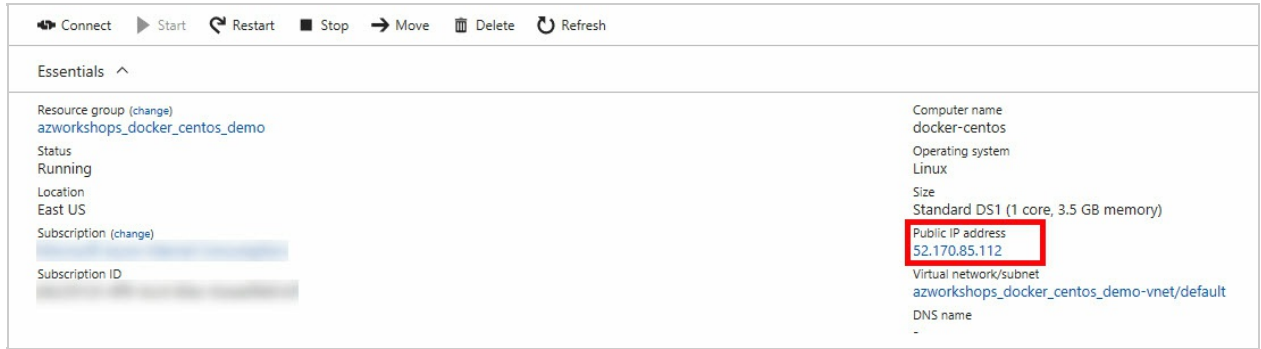
Connect to the Virtual Machine

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.

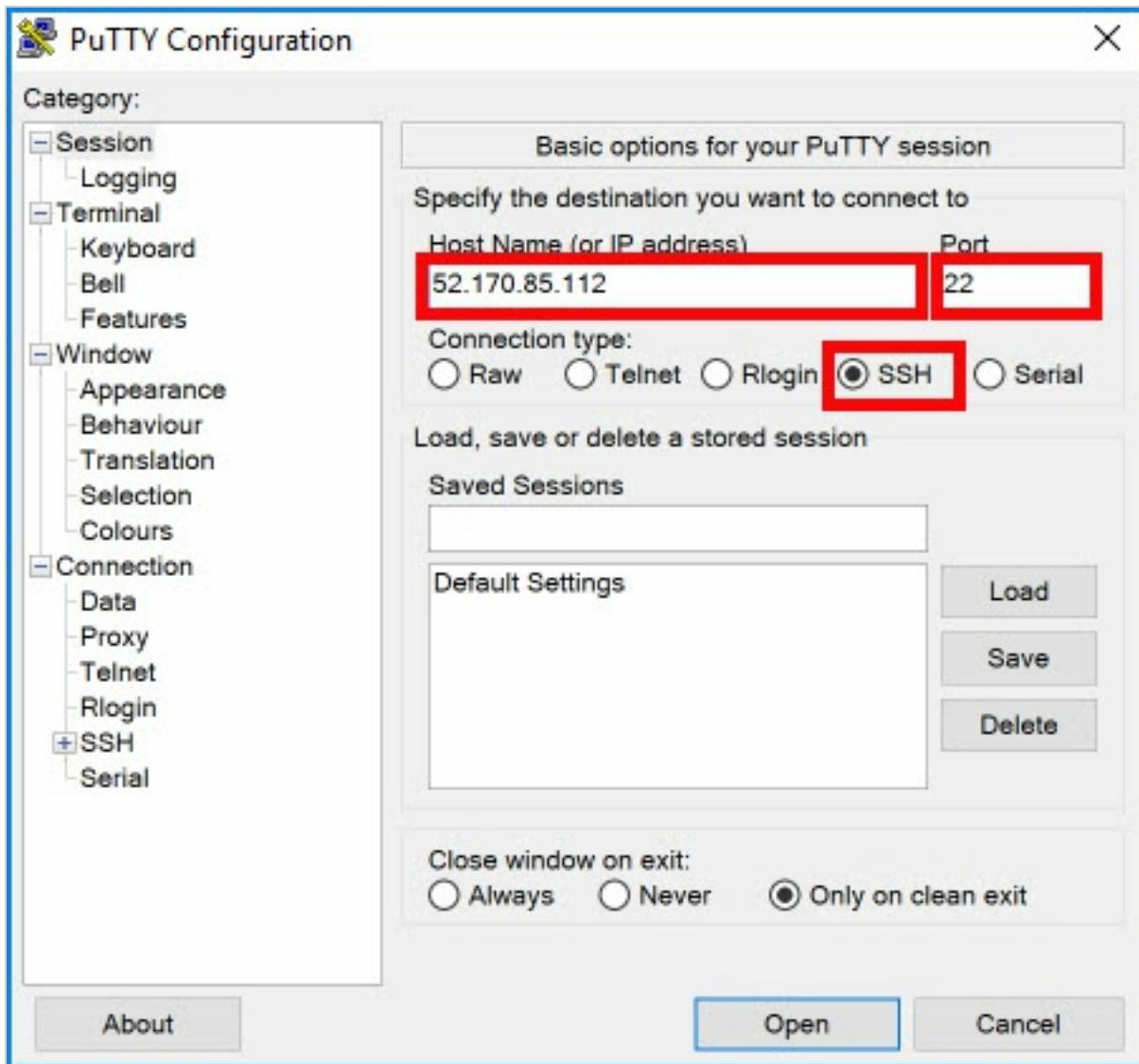
2. In the top section of the blade, in the right column, you should see a **Public IP address** listed.



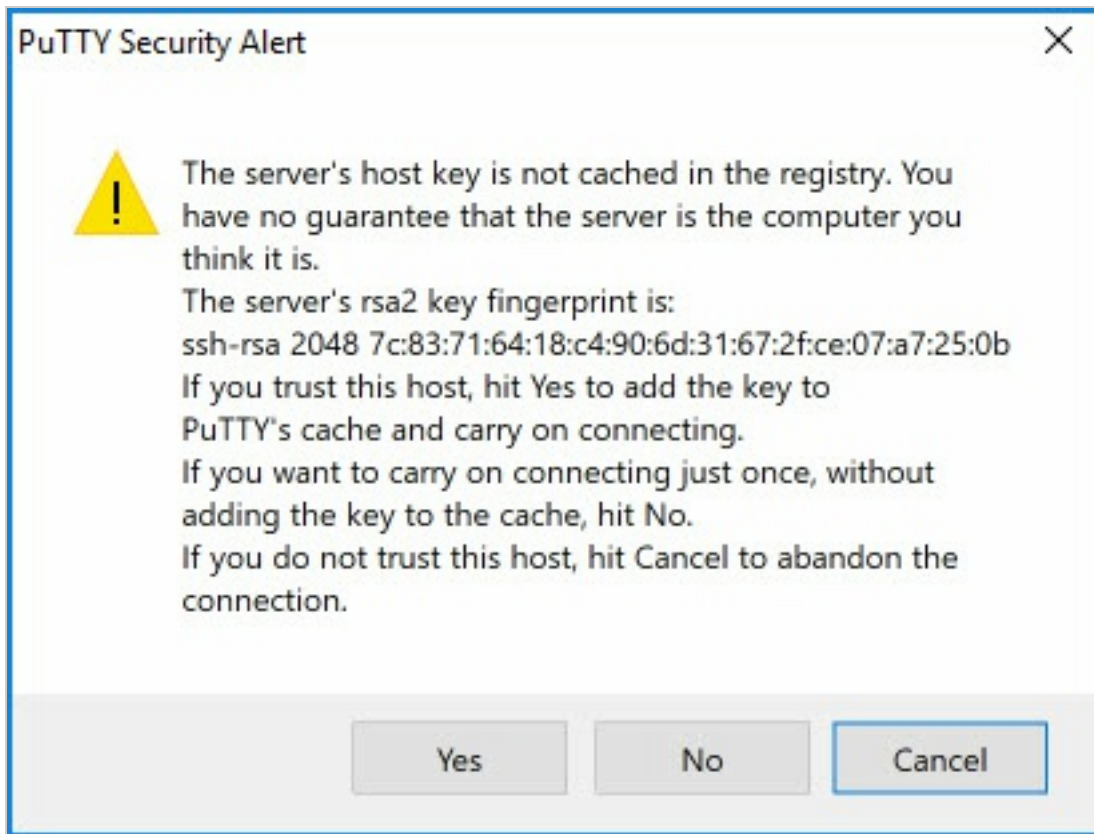
3. Copy the IP address.

Connect with SSH

1. Open **PuTTY**.
2. In the configuration window:
 - o Hostname: **<IP address from previous step>**
 - o Port: **22**
 - o Connection type: **SSH**



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



5. You will then connect to the remote CentOS 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@dockerfile-centos ~]$
```

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

Install Updates

Overview

We have just created our CentOS 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 CentOS server and login.
2. From the command prompt, type the following to automatically install all available updates:

```
sudo yum update -y
```

3. You'll be required to re-enter your password.
4. 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

Overview

In this step, we are going to install Docker. This is one of the steps that will actually *not* be performed inside of the container image. But, of course, we need Docker on the host in order to run containers.

Install Docker

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

1. First, let's remove any remnants of older versions of Docker to ensure that we run the latest version. From the command prompt, type the following:

```
sudo yum remove docker docker-common container-selinux docker-selinux docker-engine
```

Copy & Paste

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

2. We need to install dependencies for Docker:

```
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
```

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

```
sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
```

4. Now that the new repository has been added, we need to update the `yum` package index:

```
sudo yum makecache fast
```

5. We need to query for the latest version of Docker available in the repository:

```
yum list docker-ce.x86_64 --showduplicates |sort -r
```

6. You should see outputted list *similar* to the following:

docker-ce.x86_64	17.03.1.ce-1.el7.centos	docker-ce-stabl
e		
docker-ce.x86_64	17.03.0.ce-1.el7.centos	docker-ce-stabl
e		

7. The latest version of Docker will be the top line. The version we want is listed in the middle column. In this case it's `17.03.1.ce-1.el7.centos`. To install, run the following command replacing `<VERSION>` with the version listed in the center column.

```
sudo yum install -y docker-ce-<VERSION>
```

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

9. Map the storage device for Docker to use. We'll need to temporarily promote ourselves to the highest user permission level.

```
sudo bash
mkdir /etc/docker
echo '{ "storage-driver": "devicemapper" }' > /etc/docker/daemon.json
exit
```

10. Finally, start Docker:

```
sudo systemctl start docker
```

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:

```
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.service
to /usr/lib/systemd/system/docker.service.
```

Elevate Your Privileges

By 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 this 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.

Install Node.js

Overview

Now that we have the host machine configured for Docker, we're going to begin building our web server. Keep in mind that this virtual machine is serving two purposes: 1) our Docker host; and, 2) our temporary web server until our image is built.

Install Node.js

Node.js is a JavaScript execution engine that allows us to write server-side JavaScript and utilize it like any other executable file. With Node.js, we will host a very simple web server.

Add EPEL Repository

One installation method for installing Node.js uses the EPEL (Extra Packages for Enterprise Linux) repository that is available for CentOS and related distributions.

To gain access to the EPEL repo, you must modify the repo-list of your installation. Fortunately, we can reconfigure access to this repository by installing a package available in our current repos called `epel-release`.

```
sudo yum install -y epel-release
```

Download and Install Node.js Runtime

The Node.js runtime is what is responsible for hosting and executing our JavaScript.

From the terminal prompt, type:

```
sudo yum install -y nodejs
```

Download and Install NPM

NPM is an abbreviation for *Node Package Manager*. NPM allows us to install Node.js dependencies for our applications.

At the prompt, type:

```
sudo yum install -y npm
```

Verify Installation

Let's make sure our installation was successful.

At the prompt, type:

```
node -v  
npm -v
```

You should receive version numbers for both commands.

We've now successfully completed the installation of Node.js and the Node Package Manager.

Download Sample Website

Overview

We've downloaded most of our dependencies. In this step we will download a demo web site from GitHub, a remote source code repository.

Clone the Website

'Cloning' is the method in which you download a remote repository to your local machine. We are going to clone the demo web site to the default web hosting folder on our system.

Install Git

Git provides a command-line interface (CLI) for interacting with Git repositories. We need to install Git on our CentOS machine.

```
sudo yum install -y git
```

Download the Website

From the command prompt, type the following:

```
sudo git clone https://github.com/AzureWorkshops/samples-simple-nodejs-website.  
git /var/www
```

You should see something similar to the following:


```
Cloning into '/var/www'...
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 9 (delta 0), reused 6 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
Checking connectivity... done.
```

The website source code has now been downloaded.

Download Dependencies

Overview

Even though we've installed *system* dependencies, our website has a couple of library dependencies that we need to install. We'll also ensure that the website is functioning correctly, before we proceed to build our Dockerfile.

Download Website Dependencies

We'll need to enter into the website's main folder to install the dependencies.

Type the following into the terminal:

```
cd /var/www  
sudo npm i
```

You should see a number of progress bars displayed as dependencies are downloaded followed by what looks like a folder hierarchy.

Test Our Website

The first thing we need to do is run our web hosting server which is handled by Node.js. If we ran this normally from the command-line, it would block all other input. We would then have to open a second terminal to test our website. Instead, we're going to run our web server in the background as a detached process.

At the terminal prompt, type:

```
sudo node index.js &
```

NOTE: The ampersand at the end of the line instructs the system to run the preceding command in the background. When we construct our Dockerfile, we will omit the ampersand so that the process runs in the foreground and creates a long-running process that keeps our container 'alive'.

When you run this command, a number is outputted to the screen. This number refers to a *process id* (pid). Keep this number handy as we'll use it below to 'kill' the background process.

Additionally, you should see a message stating that our 'server is listening on 8080'.

Test that the web server is returning our site by typing the following:

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

If all is successful, you should receive some HTML source code back.

Stop the Background Process

We don't need the website running any longer now that we know it works. Additionally, if we kept it running, it's use of the port 8080 could create potential conflicts if another service (e.g. container) is needing that port.

Referring to the process id (pid) above, execute the following command with your id:

```
sudo kill <pid>
```

If successful, you should see the following confirmation message:

```
[1]+  Exit 143          sudo node index.js
```

Building our web server has been successful. We are now ready to move forward and replicate our work in building out an image.

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 CentOS
2. Install the latest packages
3. Install and configure Docker
4. Add a reference to EPEL
5. Install Node.js
6. Install Node Package Manager (NPM)
7. Install Git
8. Download (clone) the sample website
9. Download website dependencies
10. Run the web server

As a reminder, since we are constructing an image, we can **ignore** step 3. We won't need Docker installed inside of the image.

Create the Dockerfile

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

1. Return to your home folder by typing at the terminal prompt, `cd ~`.
2. Create a Dockerfile using the `nano` text editor by typing the following: `nano Dockerfile`. Nano is reminiscent of the old DOS editor. Of course, you can use `vim` instead if you are comfortable in doing so.
NOTE 'Dockerfile' is case-sensitive.
3. Enter the following without the line numbers. The line numbers are provided for reference below.

```
1 FROM centos:latest
2 MAINTAINER Your Name <you@yourcompany.com>
3
4 RUN yum update -y
5 RUN yum install -y epel-release
6 RUN yum install -y nodejs
7 RUN yum install -y npm
8 RUN yum install -y git
9
10 RUN git clone https://github.com/AzureWorkshops/samples-simple-nodejs-website.git /var/www
11 WORKDIR /var/www
12 RUN npm i
13
14 EXPOSE 8080
15
16 CMD node /var/www/index.js
```

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

5. To exit, **Ctrl+X**

Explanation

First, if you remember from the previous steps, we prepended each command with `sudo` to allow the command to be executed with elevated privileges. By default, all Docker images execute under the identity of the built-in superuser account `root`. Therefore, we can omit the `sudo`.

Line 1: Specifies the base image, including the tag, with which we're starting. In our case, we are using the minimal CentOS OS as the base image.

Line 2: Specifies the owner of the image with their email address.

Lines 4-8: The commands we executed earlier in this workshop that update the system and install Node.js and Git.

Line 10: Downloads (clones) the sample website into the `/var/www` local folder.

Line 11: `WORKDIR` is how you change the current directory (compared to `cd`) in a Dockerfile. We are changing to the website home directory.

Line 12: Install the website's dependencies.

Line 14: Our website server is programmed to use port 8080. 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.

Line 16: This starts our web server. We could have used the `RUN` directive, but the `CMD` directive is designed to execute our long-running process. While, technically, we could have multiple `CMD` lines in the Dockerfile, the Docker build will ignore all `CMD` lines except the last one.

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 command prompt, type `cd ~` to ensure you are in your home folder, then type the following:

```
docker build -t test/simpleweb .
```

This will build an image using `test/simpleweb` as the repository name. The period at the end specifies the path where Docker can find the Dockerfile.

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
<code>test/simpleweb</code> 469 MB	latest	d2c9a502fd30	3 minutes ago
centos 193 MB	latest	3bee3060bfc8	37 hours ago

Our image has been built using the specified repository name. You'll also notice that the `centos` image has been downloaded. This is because the build process required CentOS in order to build our image. Now that our image has been built, you could delete the `centos` image if you wanted to. Finally, when looking at the image sizes, you'll see that our image is 4 times larger due installation of Node.js, Git, and the other dependencies. In the end, however, 500MB is still not that large.

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

We've started 3 separated instances of our web server. We've bound the web server's internal port 8080 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	"/bin/sh -c 'node ...'"	3 seconds ago
Up 2 seconds	0.0.0.0:8082->8080/tcp	web_8082	
323a65fa5143	test/simpleweb	"/bin/sh -c 'node ...'"	11 seconds ago
Up 10 seconds	0.0.0.0:8081->8080/tcp	web_8081	
7d4fee5c8f89	test/simpleweb	"/bin/sh -c 'node ...'"	About a minute ago
Up 59 seconds	0.0.0.0:8080->8080/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 are left with successfully creating three container instances running our custom image.

Expose Site 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.

Network Security Group (NSG)

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

When we created our CentOS 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 our three ports (8080-8082) so that we can access all three containers.

1. If you are not still there, go back to the Azure portal and navigate to the settings of your CentOS virtual machine.
2. In the left menu, click on **Network interfaces**  .
3. This will open the *Network Interfaces* blade for your CentOS 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 - **dockerfile-centos-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:
 - o Name: **allow-http**
 - o Priority: **1010**
 - o Source: **Any**
 - o Service: **Custom**
 - o Protocol: **Any**

- Port range: **8080-8082**
- 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 CentOS virtual machine, including the port number. The IP address used in this workshop's screen shots is **52.170.85.112** (your IP address will be different). Using the aforementioned IP address, I would direct my browser to **http://52.170.85.112:8080**. I would also test the other 2 port numbers (e.g. 8081, 8082). You should see the 'Hello World' page at all three URL/port combinations.