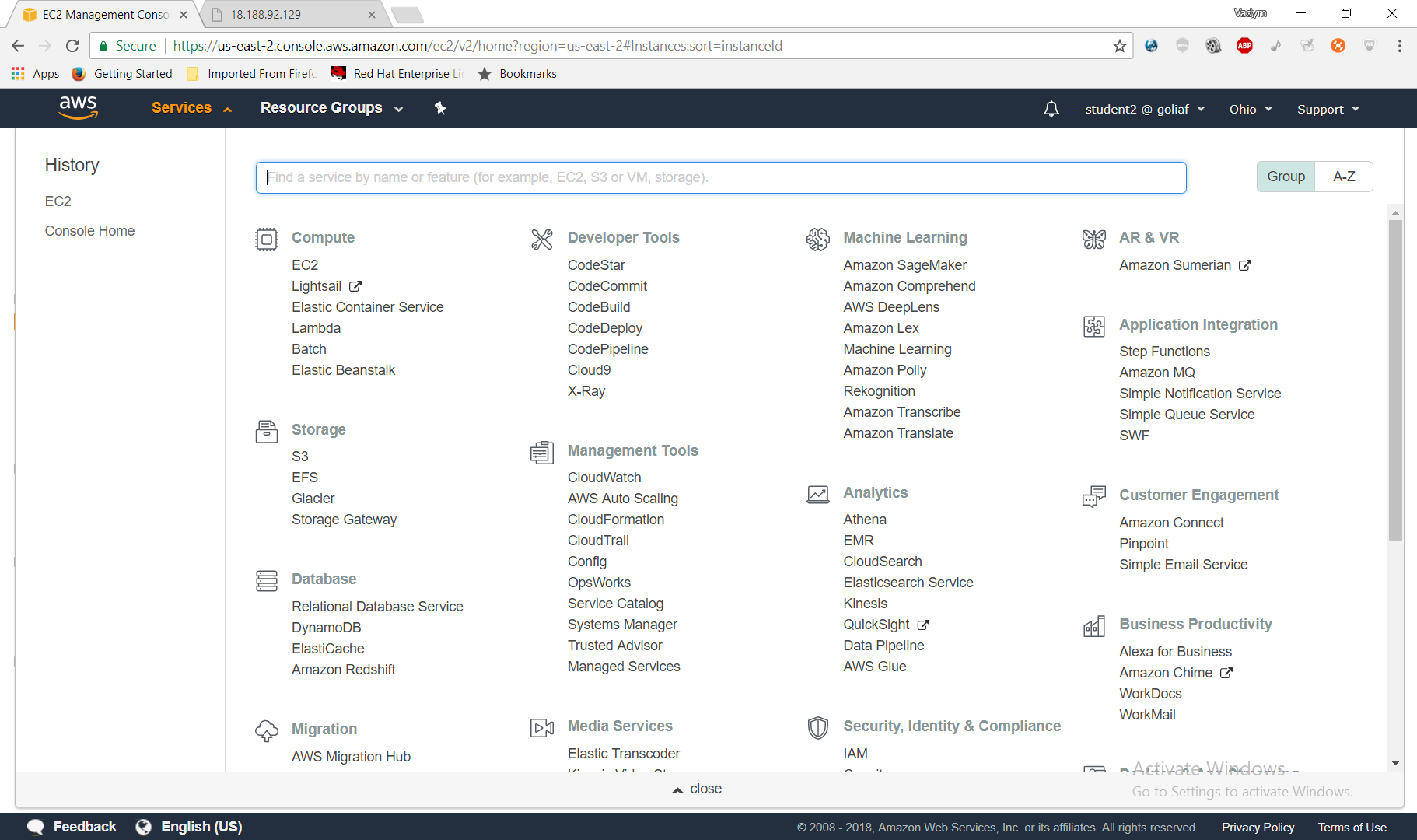
**Step 1 Logging in to the Amazon Web Services Console**

**Introduction**

This Lab experience involves Amazon Web Services, and you will use the AWS Management Console to complete all the Lab Steps. Please note that you will have a space storage limit of 20GB for this lab, which will be more than sufficient to complete it.



The AWS Management Console is a web control panel for managing all your AWS resources, from EC2 instances to SNS topics. The console enables cloud management for all aspects of the AWS account, including managing security credentials, or even setting up new IAM Users.

**Instructions**

1. To start the Lab experience, open the Amazon Console by clicking this button:

<https://goliaf.signin.aws.amazon.com/console>

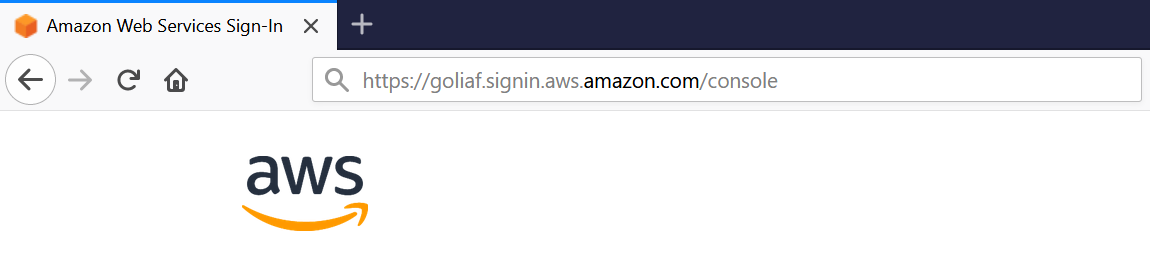
2. Enter the following credentials created just for your Lab session, and click **Sign In**:

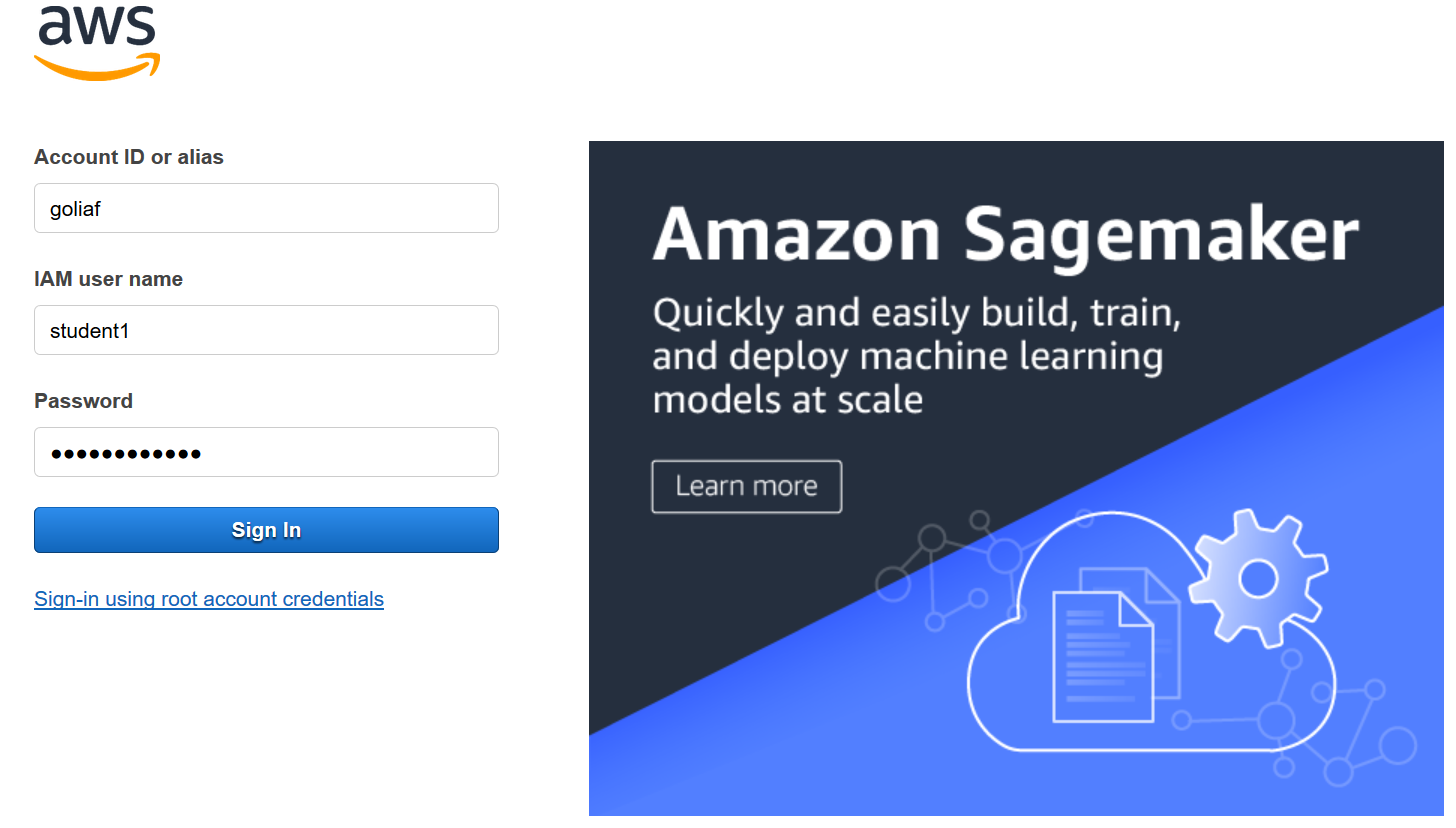
* **Account ID or alias**: Keep the pre-populated value
* **IAM user name**: *student{01-17}*
* **Password**: *$trainig0*
* **Signing-in AWS Management Console**

We are using URL

<https://goliaf.signin.aws.amazon.com/console>

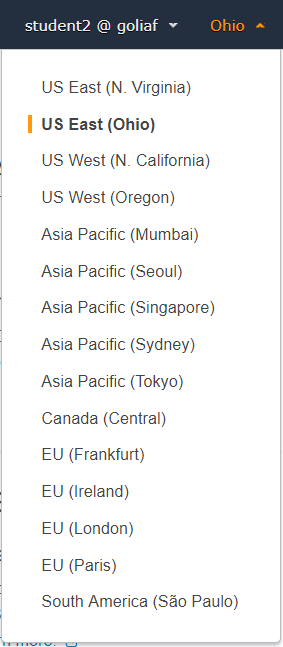
to sign-in AWS Management Console for training purpose.



After that we will be redirected to AWS Login page: 

Enter IAM User name and Password here and then click “Sign In” button. AWS Console Home page will be opened:

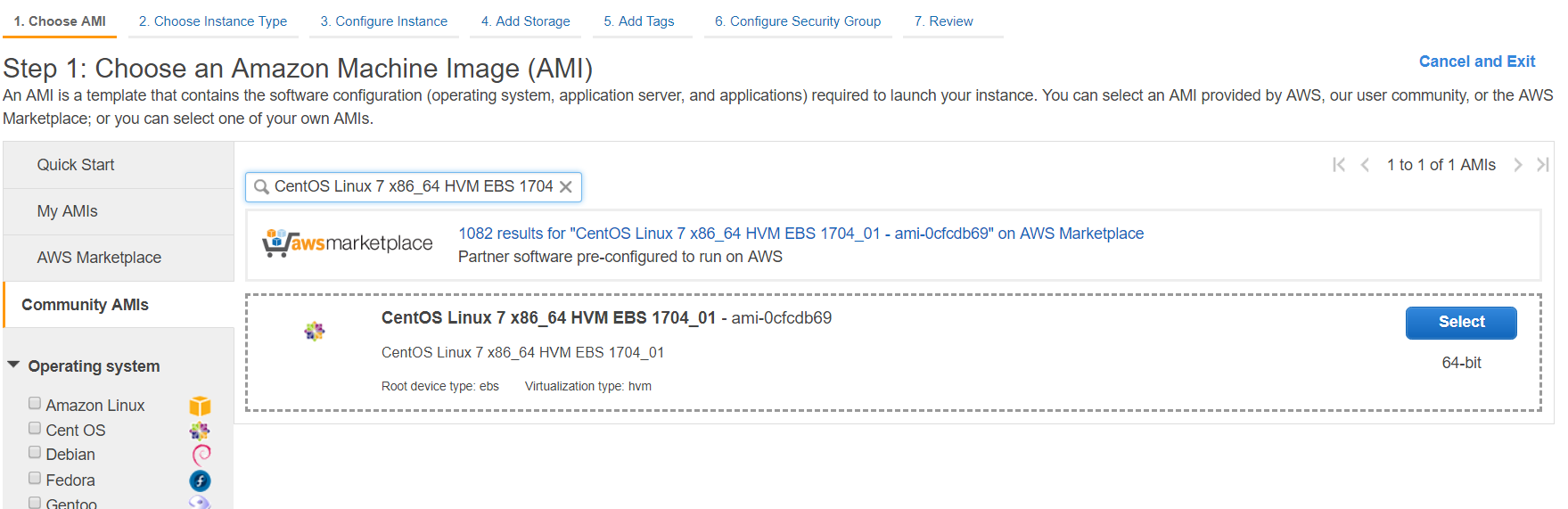
3. Select the **US East (Ohio)** region using the upper right drop-down menu on the AWS Management Console:

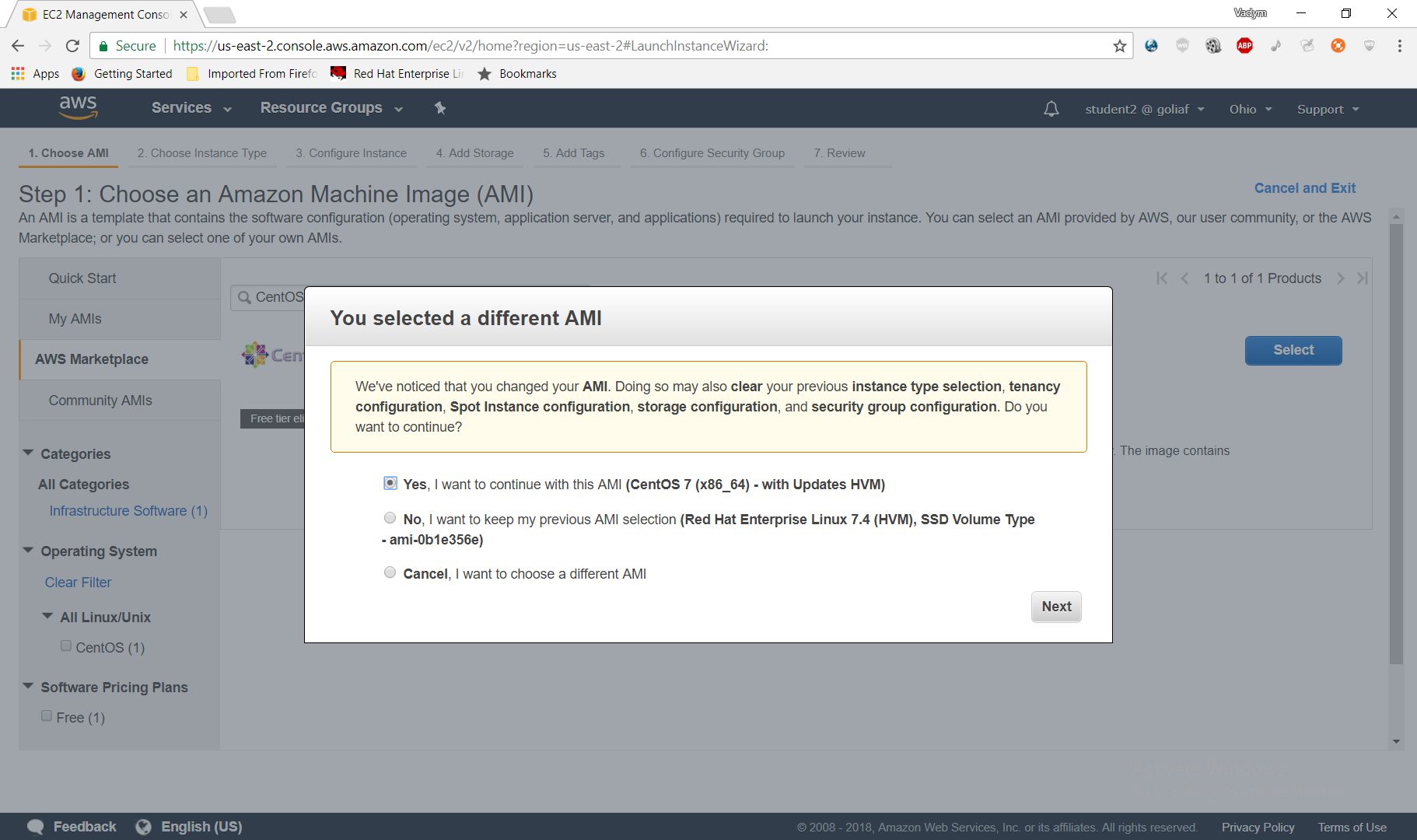


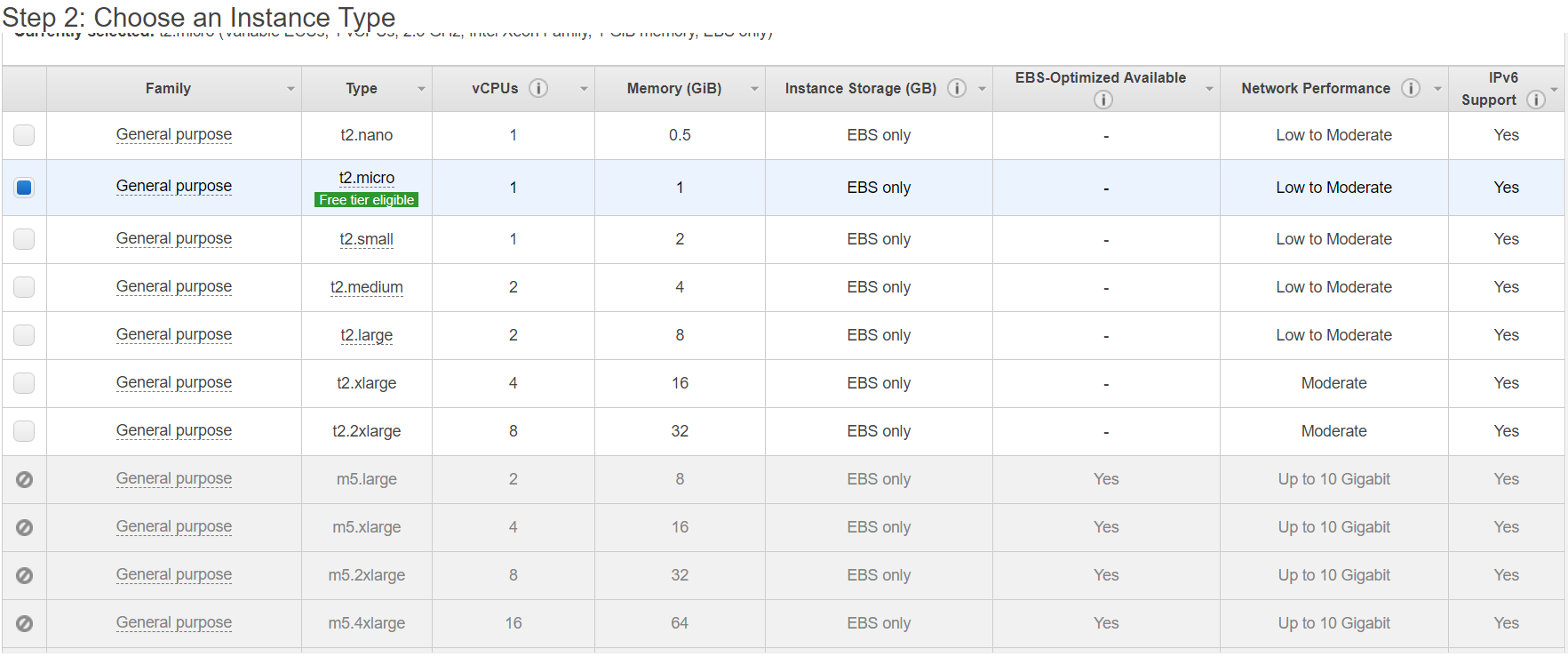
Amazon Web Services is available in different regions all over the world, and the Console lets you provision resources across multiple regions. You usually choose a region that best suits your business needs to optimize your customer’s experience, but you must use the **US East (Ohio)**for this Lab.

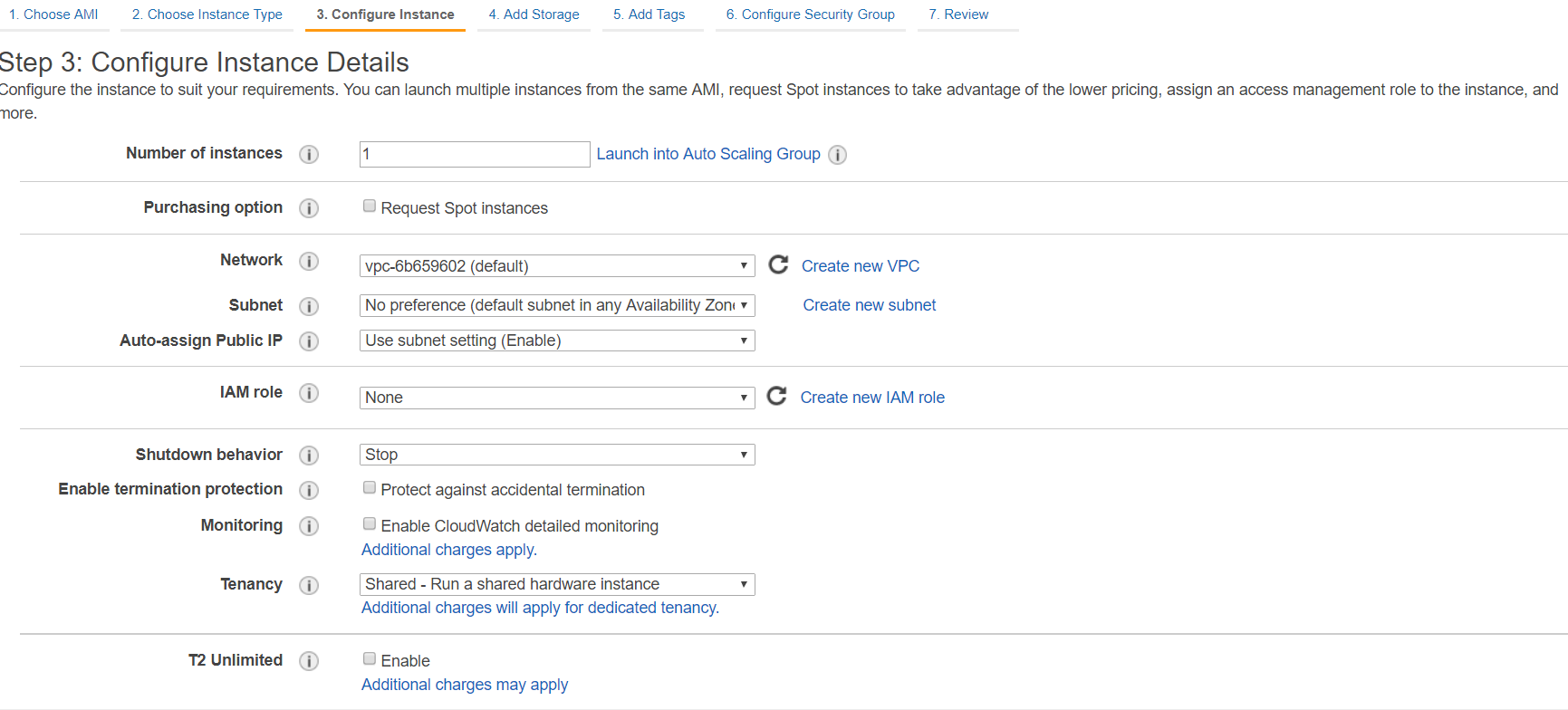
## Step 2 Launching EC2 instance

Select EC2 service and Launch instance. Choose AWS marketplace and supply “**CentOS Linux 7 x86\_64 HVM EBS 1704\_01** - ami-0cfcdb69” name for search.

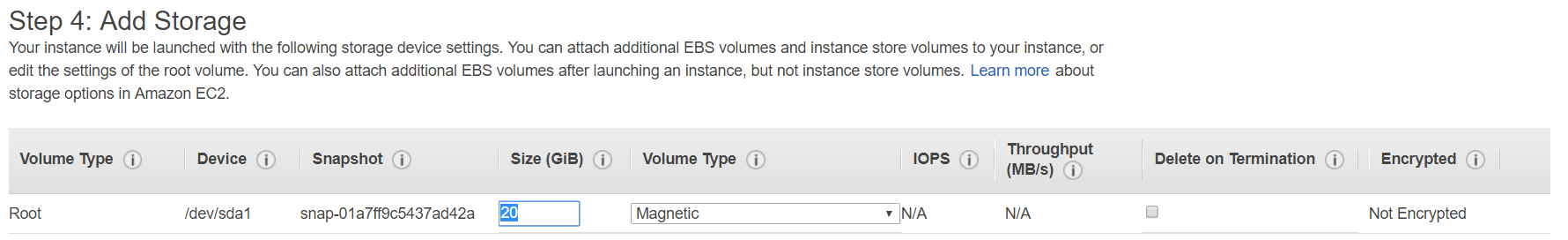




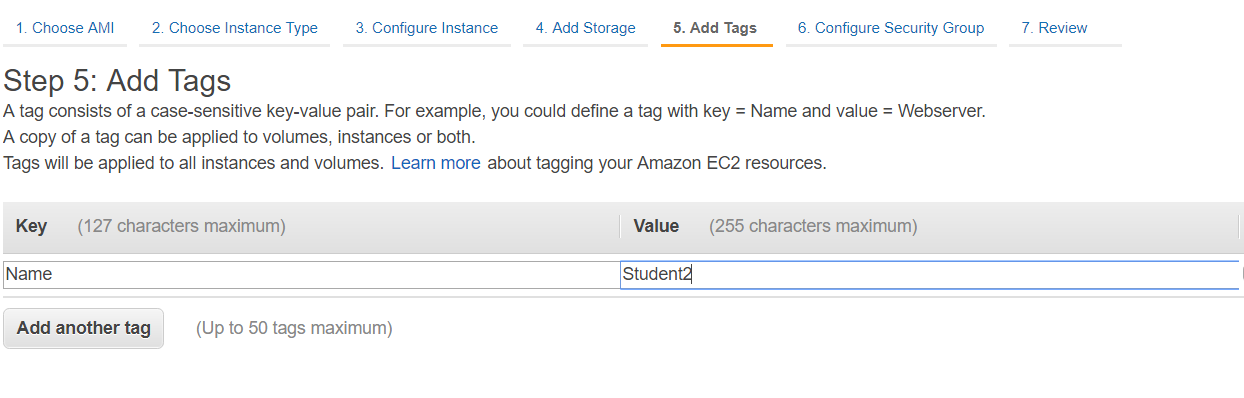


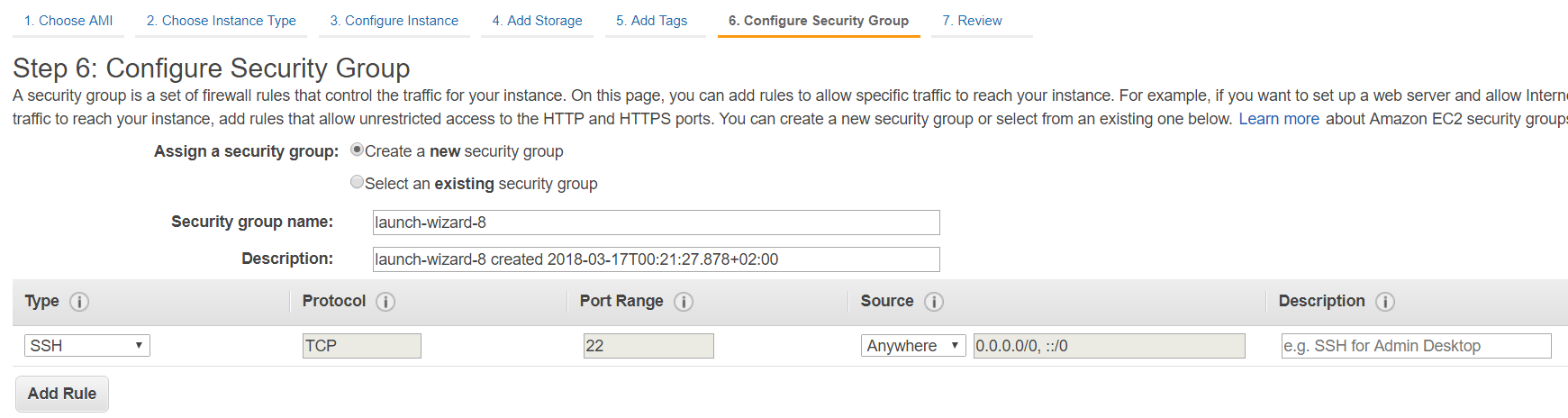


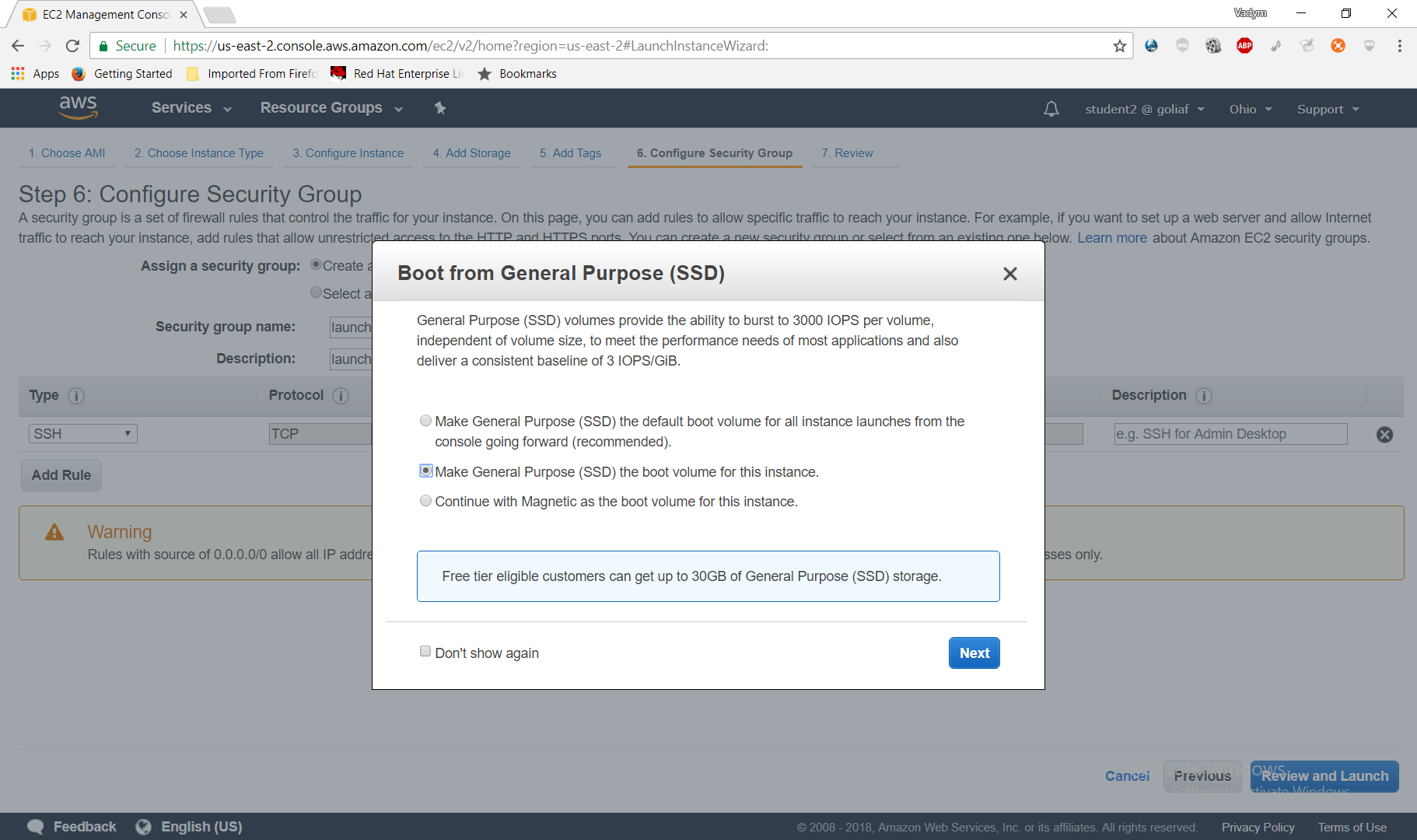
**Specify 20Gb for Root Volume**

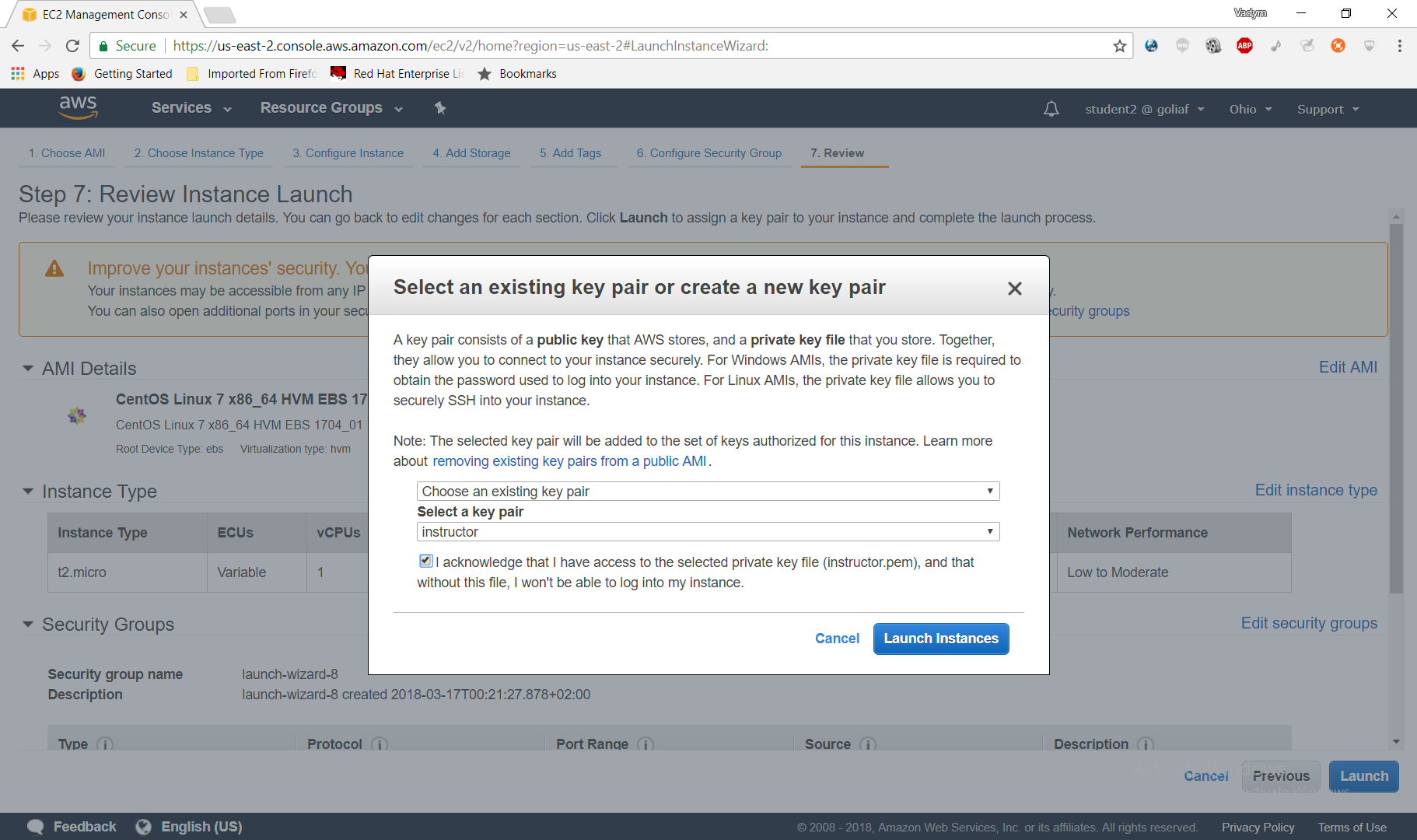


**Define your own tag**









## Step 3 Connecting to the Virtual Machine using SSH

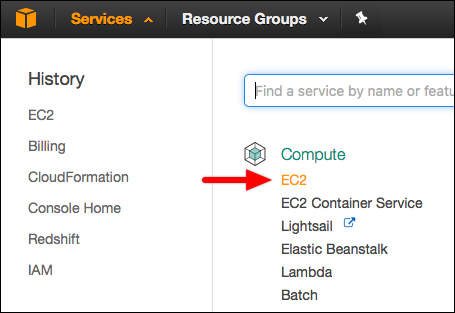
### Introduction

In this Lab Step, you will employ a Secure Shell (SSH) client to connect to a remote Linux server. SSH is a cryptographic network protocol for securing data communication. SSH establishes a secure channel over an unsecured network. Common applications include remote login and remote command execution.

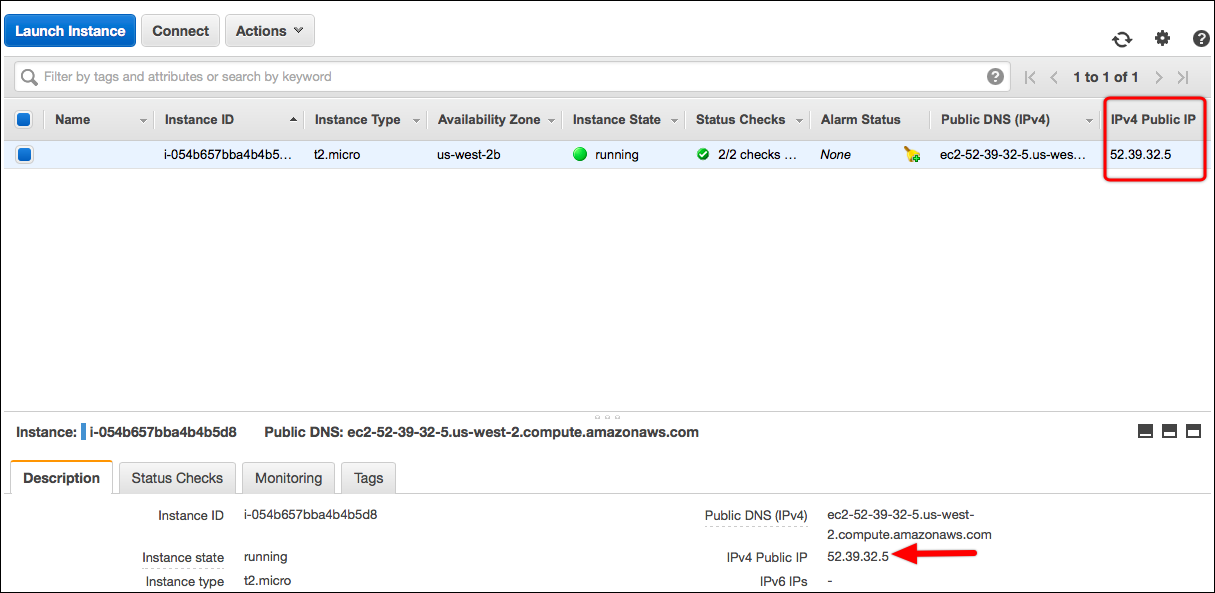
### Instructions

#### Locating the virtual machine IP address

1. In the AWS Management Console, navigate to **Services** > **Compute** > **EC2**:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid1-4657091f-f721-4522-9c3f-c72d8099f0a5.png)

2. Click on **Running Instances** and select the target virtual machine and locate the **IPv4 Public IP** address. An example IPv4 address number is 52.39.32.175. Copy your virtual machine IP address for later use:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid3-abd52ee9-37cb-4050-8e16-fb8919aaa316.png)

3. Proceed to the **Connecting using Linux / macOS** or **Connecting using Windows** instructions depending on your local operating system.

### 

#### Connecting using Linux / macOS

Linux distributions and macOS include an SSH client that accepts standard PEM keys. Complete the following steps to connect using the included terminal applications:

a. Open your terminal application. If you need assistance finding the terminal application, search for terminal using your operating system's application finder or search commands.

b. Enter the following command and press Enter:

ssh -i /Path/To/Your/KeyPair.pem AMIUserName@YourIPv4Address

where the command details are:

ssh initiates the SSH connection.

-i specifies the identity file.

/Path/To/Your/Keypair.pem specifies the location and name of your key pair. An example location might be /Home/YourUserName/Downloads/KeyPair.pem.

AMIUserName specifies the SSH user:

For the Amazon Linux Amazon Machine Image (AMI), a standard SSH user is ec2-user.  
For Ubuntu images, a standard SSH user is ubuntu.   
For CentOS images, a standard SSH user is centos.   
For Debian images, a standard SSH user is admin.  
For Red Hat 6.4 and later images, a standard SSH user is ec2-user.

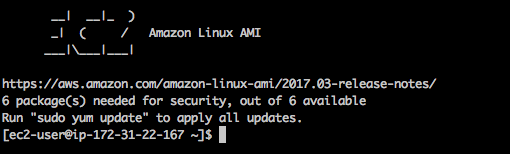
YourIPv4Address is the IPv4 address noted earlier in the instructions.

Note: Your SSH client may refuse to start the connection due to key permissions. If you receive a warning that the key pair file is unprotected, you must change the permissions. Enter the following command and try the connection command again:

chmod 600 /Path/To/Your/KeyPair.pem

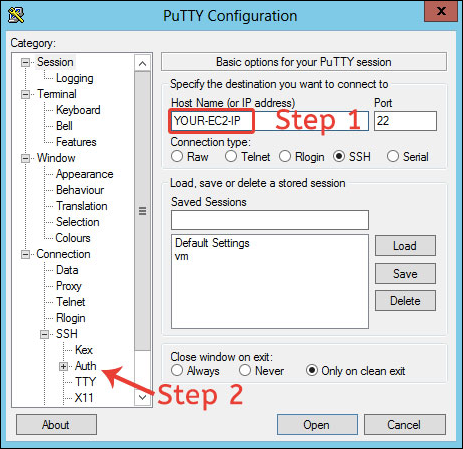
c.  After successfully connecting to the virtual machine, you should reach a terminal prompt similar to the one shown in the image below.

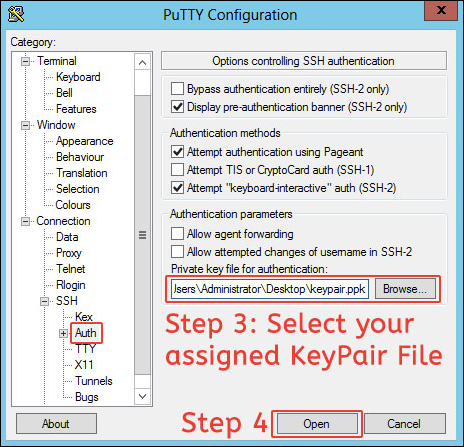
Note: If you receive a warning that the host is unknown, enter y or yes to add the host and complete the connection.

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid5-cbc3b720-f479-4142-bb16-380a2bf5a2e2.png)

#### Connecting using Windows

Windows does not include an SSH client. You must download an application that includes one. A free and useful utility is called PuTTY. PuTTY supports SSH connections as well as key generation and conversion. Download PuTTY at [http://www.putty.org](http://www.putty.org/). Complete the following steps to use PuTTY to create an SSH connection.

a. Open PuTTY and insert the IPv4 public IP address in the **Host Name (or IP address)** field.  
[](https://assets.cloudacademy.com/bakery/media/uploads/blobid8-b7b9cff0-0e27-439d-a31a-69e4a0591c30.png)

b. Navigate to the **Connection** > **SSH** > **Auth** section. Select the PPK key pair you downloaded earlier, and click **Open**.  
[](https://assets.cloudacademy.com/bakery/media/uploads/blobid11-05f54731-cc3c-487b-b4a3-1af16f4fc10b.png)

c. Wait several seconds for the authentication prompt. Enter the SSH username for the virtual machine operating system, such as ec2-user for Amazon Linux, and press Enter.

Additional example SSH usernames include:

For Amazon Linux, a standard SSH user is ec2-user.  
For Ubuntu images, a standard SSH user is ubuntu.   
For CentOS images, a standard SSH user is centos.   
For Debian images, a standard SSH user is admin.  
For Red Hat 6.4 and later images, a standard SSH user is ec2-user.

## Step 4 Installing Docker on Linux

### Introduction

You will install Docker using the yum package manager that is available on CentOS 7.3. Docker comes in two flavors: Community Edition (CE) and Enterprise Edition (EE). The community edition is open source and available free of charge. It includes the core Docker functionality and was previously called "Docker Engine." The enterprise edition requires a subscription and includes a support package, a certification program to create trusted containers and extensions, and other features. You will be installing the community edition. You will also see how to add your user to the docker group to use the Docker commands without elevating to root permissions.

### Instructions

1. In the ssh terminal, enter the following command to install the **yum-utils** package:

sudo yum install -y yum-utils

The yum-utils package includes a handy utility for adding package repositories that you will use next.

 2. Enter the following to add the Docker CE package repositories to your system:

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

 3. Enter the subsequent command to update the yum cache with the Docker repositories:

sudo yum makecache fast

 4. Enter the following to install Docker CE:

sudo yum -y install docker-ce

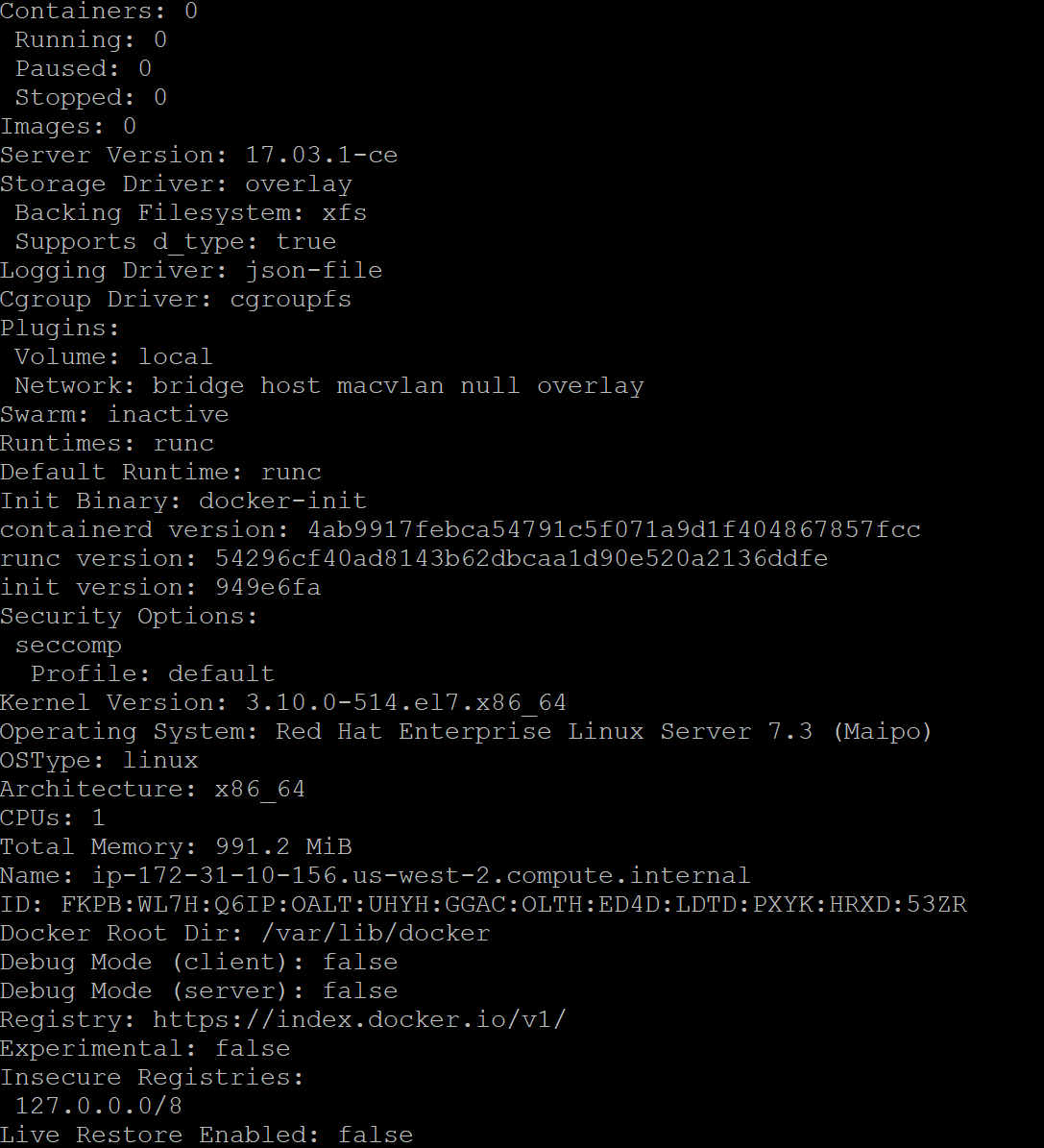
 5. Enter the command below to start Docker as a service:

sudo systemctl start docker

 6. Verify Docker is running by entering:

sudo docker info

This will output system-wide information about Docker. The information will resemble the following:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid0-d0037dbf-18c2-4e4b-aa48-a1e28a3e5fb5.png)

You can see some useful information, such as the number of containers, and the version of the Docker server. Docker adopts a client-server architecture, so the server doesn't have to be running on the same host as the client. In your case, you are using the Docker command line interface (CLI) client to connect to the server, called the Docker daemon.

## Step 5 Using Docker without Root Permission on Linux

### Introduction

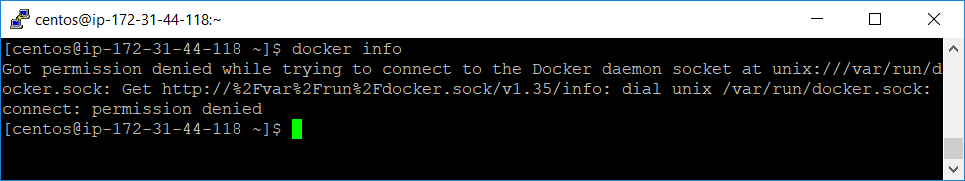
In the last Lab Step, you used root permissions to run a Docker command. In this Lab Step you will learn how to use Docker commands without elevating to root permissions. This may not always be what you want and should only be performed for trusted users. This is because the actions are equivalent to granting root permissions to a user. It will be convenient in the following Lab Steps to not have to enter sudo for every command.

### Instructions

1. Try entering the same command without using root permission:

docker info

You will receive a permission denied error message similar to:



By default, the Docker daemon will reject requests from users that aren't part of the **docker** group. If you encounter this message in your travels, you can either use root permission or add your user to the docker group.

 2. Verify the docker group exists by searching for it in the groups file:

grep docker /etc/group

If you don't see a line beginning with "docker:", you will need to add the group yourself by entering:

sudo groupadd docker

3. Add your user to the docker group:

sudo gpasswd -a $USER docker

The groups of the currently logged in user is cached, you can verify this by entering groups.

 4. You can login again to have your groups updated by entering:

newgrp docker

Now you will have docker in your list of groups if you enter groups.

 Note: It is convenient to not have to terminate your current ssh session by using newgrp, but terminating the ssh session and logging in again will work just as well.

 5. Verify that your user can successfully issue Docker commands by entering:

docker info

 Note: if you don't see the system-wide Docker information, you may need to restart the Docker daemon by entering sudo systemctl restart docker.

## Step 6 Getting Docker Help from the Command Line

### Introduction

In this Lab Step, you will learn about how Docker commands are organized and how to get help for commands on the command line. If you are familiar with using the Linux command line or the git version control system, you will find many of the commands familiar in name and action. Even if you aren't familiar, the commands are organized in an intuitive way that makes learning to use Docker commands as painless as practicable.

The commands are organized into common commands and a more exhaustive list grouped around the management of a specific component of Docker. You use each with a different syntax. For a common command, the usage is:

docker command-name [options]

and the usage for a management group command is:

docker management-group command-name [options]

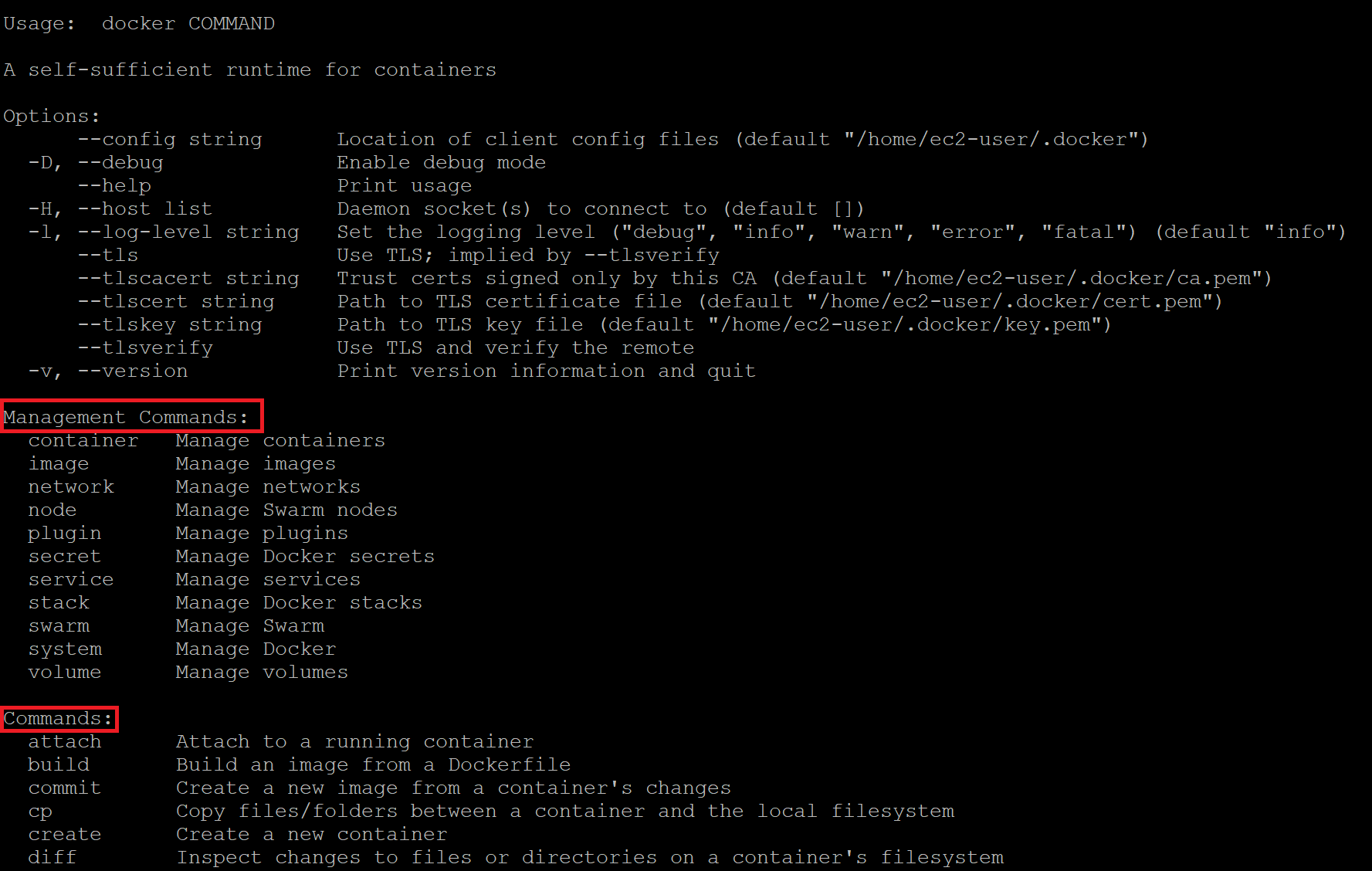
Most of the common commands can be accessed in the short form or by specifying the management grouping they also fall under. Some commands are only available by specifying the management group. You will get the hang of this and review a couple of the main concepts in the following instructions.

### Instructions

1. To see a list of the commands in Docker, simply enter:

docker --help

The management commands and common commands are highlighted in the image below:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid0-2710d102-32d2-45f5-b2e6-4a0c245517f2.png)

2. Enter the following management command:

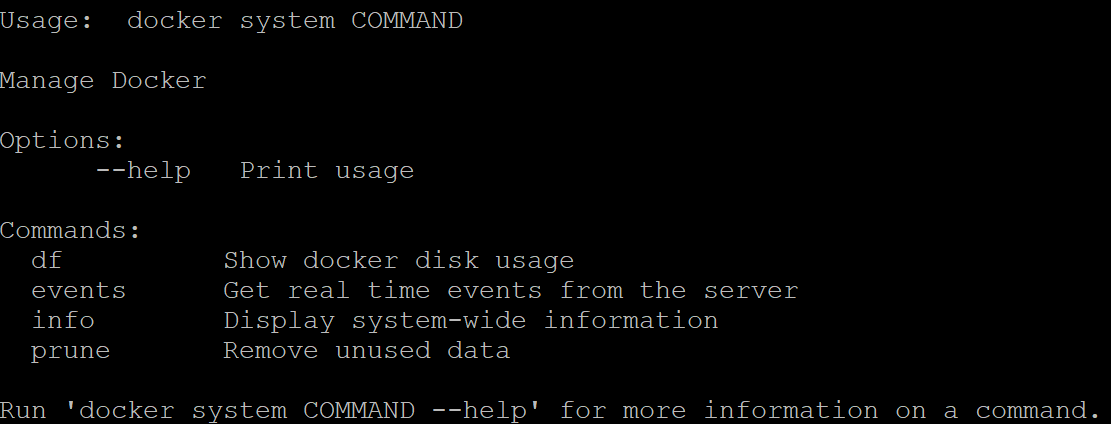
docker system info

The output should look familiar. That is because info is a common command that you used in the last Lab Step (docker info) to see the same system-wide information.

3. To see all of the commands grouped under system, enter:

docker system --help

You will be presented with output similar to:

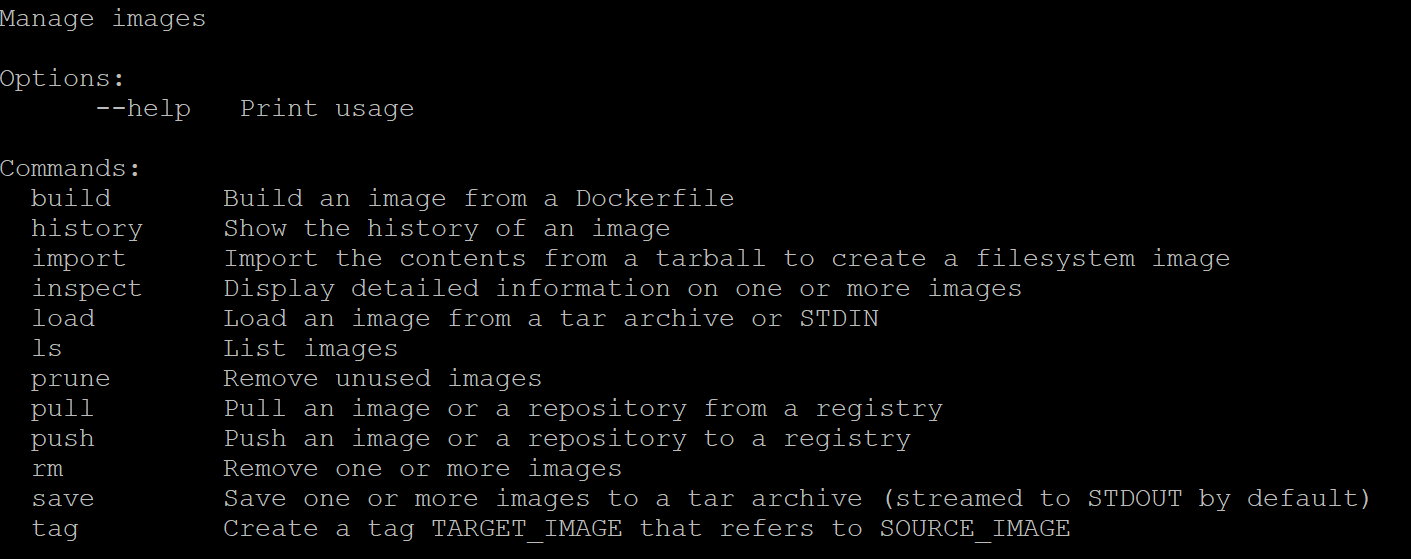
[](https://assets.cloudacademy.com/bakery/media/uploads/blobid1-a12bdbef-e19d-4149-8094-d3db275fceb8.png)

Notice the info command previously entered.  events is a common command, but df and prune are only available through the system management command.  You can use df to see the disk usage for Docker and prune to clean up unused data.

Using the --help you can explore everything there is to know about Docker commands. To prepare for the upcoming Lab Steps, you will now focus in on a few important components.

4. To view the commands grouped with images, enter:

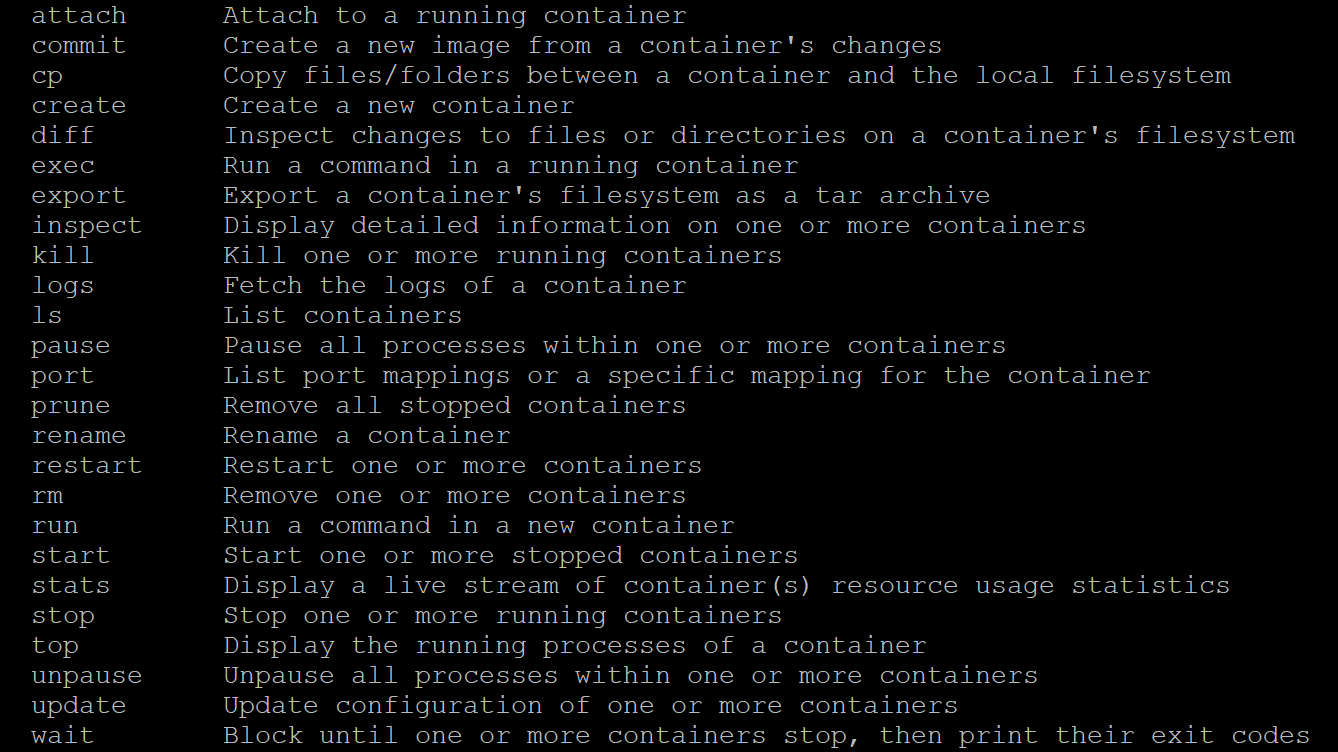
docker image --help

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid2-b261bd9e-c894-4d62-b125-ca669db5943c.png)

Images are read-only snapshots that containers can be created from. Images are built up in layers, one image built on top of another. Because each layer is read-only, they can be identified with cryptographic hash values computed from the bytes of the data. Layers can be shared between images as another benefit of being read-only. You can, and will later, build your own images. The build command accomplishes that. When you build your own image, you will select a base image to build on top of with your custom application. A pull can be used to pull or download an image to your server from an image registry, while push can upload an image to a registry. Read through the other command descriptions so you are aware of what else is available.

 5. To view the commands grouped with containers, enter:

docker container --help

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid3-33f020a0-2a22-44d7-baa1-936ae430c036.png)

A container is another core concept in Docker. Containers run applications or services, almost always just one per container. Containers run on top of an image. In terms of storage, a container is like another layer on top of the image, but the layer is writable instead of read-only. You can have many containers using the same image. Each will use the same read-only image and have its own writable layer on top. Containers can be used to create images using the commit command, essentially converting the writable layer to a read-only layer in an image.

The relationship between images and containers aside, run is used to run a command on top of an image. A container can be stopped and started again. ls is used to list containers. It is aliased to ps and list as well. Read through the other commands in the list to see what else is available for working with containers.

**Step 7 Running Your First Docker Container**

**Introduction**

As you learned in the previous Lab Step, containers run a command on top of an image. There are many available images to choose from. Images are collected in repositories. A repository can have many versions of an image. Tags are used to manage versions. You will use images from repositories inside the default Docker Registry, *Docker Hub*. Docker Hub hosts official images as well as community-contributed images. Docker Hub offers free and paid accounts. You get an unlimited amount of public repositories and one free private repository with the free account.

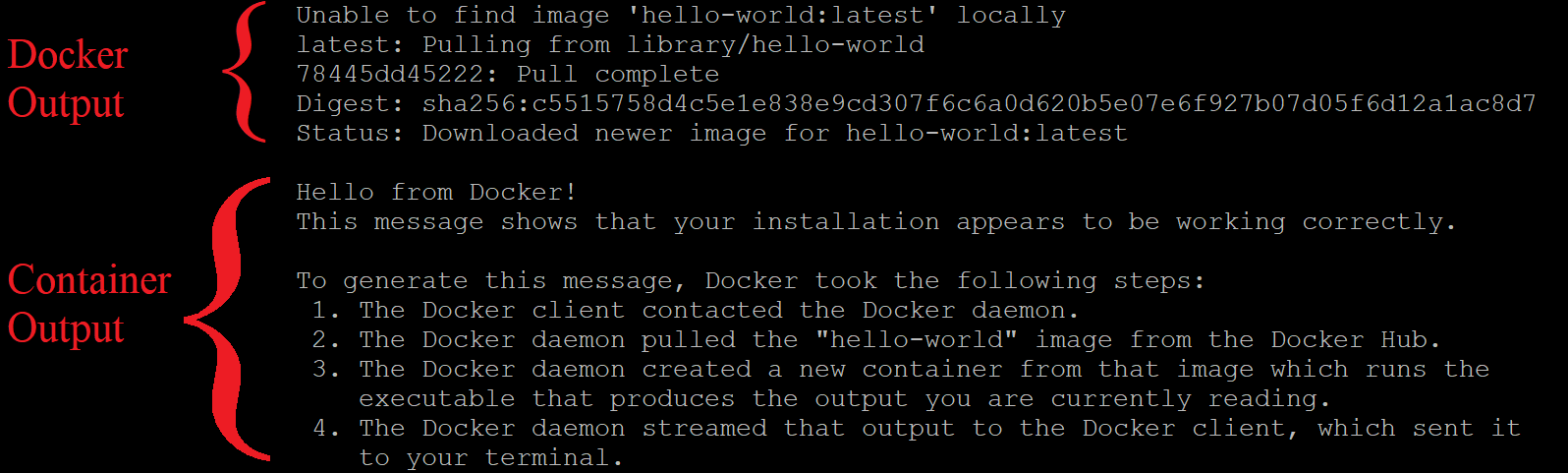
In this Lab Step, you will run your first container. You will learn how to search Docker Hub, pull images, and work with containers running on top of the images. You will get practice with many common commands when working with Docker.

**Instructions**

1. Enter the following to see how easy it is to get a container running:

docker run hello-world

It can be just that easy to run a container! You will see output similar to:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid0-04721581-c2d0-469d-9758-faeadb27a7a1.png)

There are two sections to the output: output from Docker as it prepares to run the container and the output from the command running in the container.

Take a look at the Docker output first:

1. In the first line, Docker is telling you that it couldn't find the image you specified, hello-world, on the Docker Daemon's local host. The *latest* portion after the colon (:) is a tag. The tag identifies which version of the image to use. By default, it looks for the *latest* version.
2. In the next line, it notifies you that it automatically pulled the image. You could manually perform that task using the command docker pull hello-world. The *library/hello-world* is the repository it's pulling from inside the Docker Hub registry. *library*is the account name for official Docker images. In general, images will come from repositories identified using the pattern *account/repository*.
3. The last three lines confirm the pull completed and the image has been downloaded.

The container output is the output that gets written by the command that runs in the container. Read the output to learn more about how it was produced. You ask, but what was the command? You only specified the image name, hello-world. The image provides a default command in this case which prints the output you see.

 2. Re-run the same command:

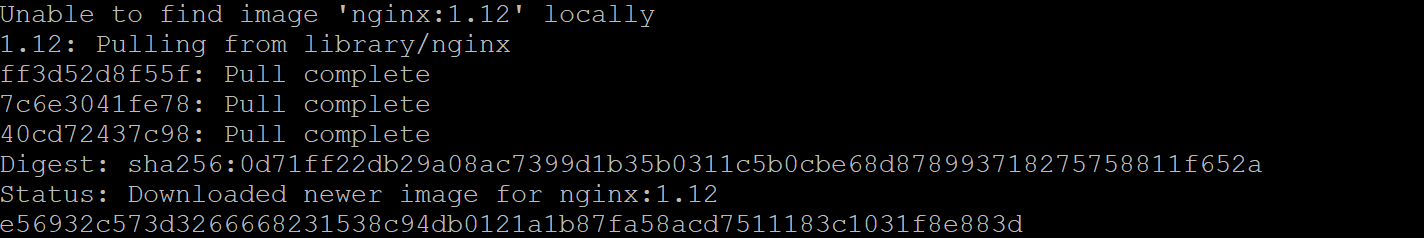
docker run hello-world

Notice this time the Docker output is not included. That is because the specific version of the image was found locally and there is no need to pull the image again.

 3. Try running a more complex container with some options specified:

docker run --name web-server -d -p 8080:80 nginx:1.12

This runs the nginx web server in a container using the official nginx image. You will see output similar to:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid4-eb9d643d-d7f4-4948-b7a8-798ad6947a99.png)

This time you specified the tag 1.12 indicating you want version 1.12 of nginx instead of the default latest version. There are three **Pull complete** messages this time, indicating the image has three layers. The last line is the id of the running container. The meanings of the command options are:

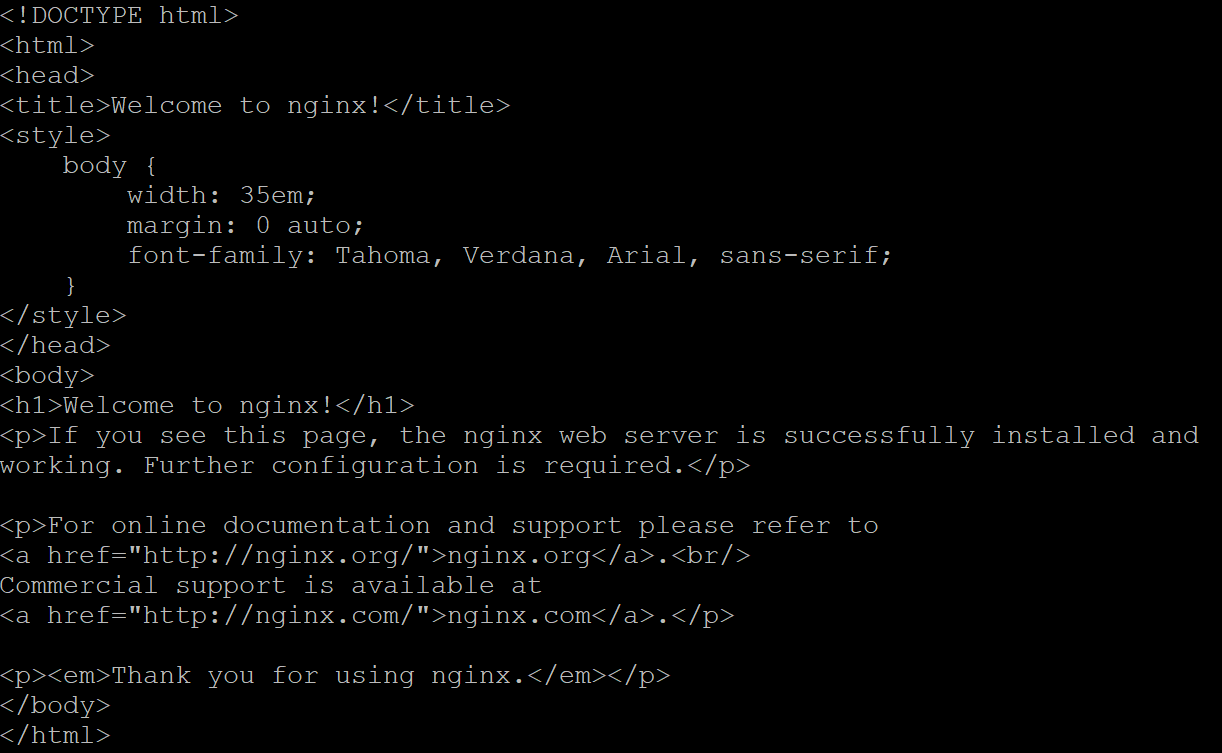
* --name container\_name: Label the container *container\_name*. In the command above, the container is labeled *web-server*. This is much more manageable than the id, 31f2b6715... in the output above.
* -d: *Detach*the container by running it in the background and print its container id. Without this, the shell would be attached to the running container command and you wouldn't have the shell returned to you to enter more commands.
* -p host\_port:container\_port: Publish the container's port number *container\_port*to the host's port number *host\_port*. This connects the host's port 8080 to the container port 80 (http) in the nginx command.

You again used the default command in the image, which runs the web server in this case.

 4. Verify the web server is running and accessible on the host port of 8080:

curl localhost:8080

This command sends an HTTP GET request (a standard web browser request) to localhost port 8080. You will be returned an HTML document, which is the default nginx web page, verifying the nginx server is running in the container:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid2-2a54382d-ff46-43cc-862e-3e606faf7cd6.png)

5. To list all running containers, enter:

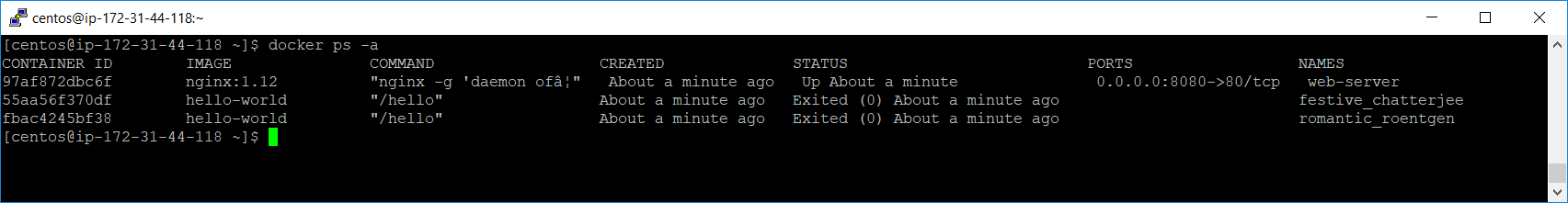
docker ps

[https://assets.cloudacademy.com/bakery/media/uploads/blobid0-5990f700-bf1c-4f14-a72e-bd3f32851183.png](https://assets.cloudacademy.com/bakery/media/uploads/blobid0-5990f700-bf1c-4f14-a72e-bd3f32851183.png)

You will see a table with one row below the table column headings, indicating one running process. The (truncated) container id is the same as the last line of the docker run output. One new thing is that you can see the command that is running in the container in the third column. The friendly name you specified is in the last column.

 6. Enter the following to see a list of all running and stopped containers:

docker ps -a



This time you can see the two hello-world containers from the start of the Lab Step. They simply wrote a message and then stopped when the command finished, whereas the nginx server is always listening for requests until you stop it. Notice that Docker automatically assigned random friendly names for the hello-world containers, *musing\_volard*, and *jovial\_snyder*in the image above. These names are useful if you need to reference a container that you didn't assign a name to by yourself.

 7. To stop the nginx server, enter:

docker stop web-server

 8. Verify the server is no longer running by running:

docker ps

This will not return any rows in the table. You can also verify that the default nginx web page is no longer being served by running curl localhost:8080 again.

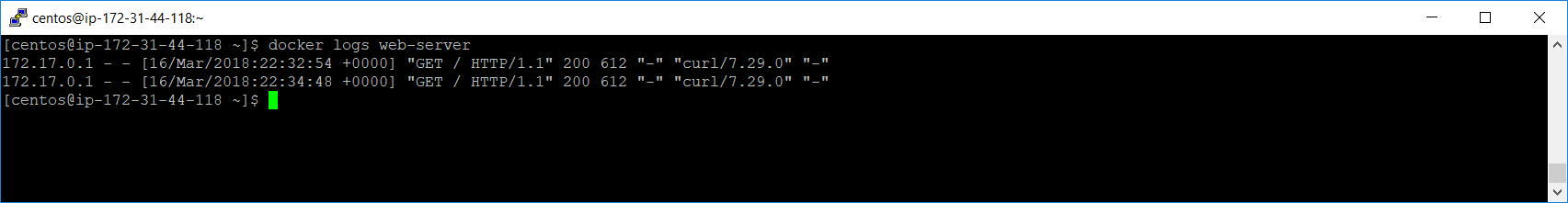
 9. To start running the command in the web-server container again, enter:

docker start web-server

This is different from re-running the original docker run command, which would make a second container running the same command instead of using the stopped container.

 10. To see the container's output messages, enter:

docker logs web-server



Docker logs messages written to standard output and standard error. In the case of nginx, it writes a line for each request that it receives.

 11. You can run other commands in a running container. For example, to get a bash shell in the container enter:

docker exec -it web-server /bin/bash

Which will cause your shell prompt to change to something similar to:

[https://assets.cloudacademy.com/bakery/media/uploads/blobid1-07c6b708-2a8a-47f1-b4df-bb468d0003de.png](https://assets.cloudacademy.com/bakery/media/uploads/blobid1-07c6b708-2a8a-47f1-b4df-bb468d0003de.png)

This indicates you are at a shell prompt in the container using the root container user. The -it options tell Docker to handle your keyboard events in the container. Enter some commands to inspect the container environment, such as ls and cat /etc/nginx/nginx.conf. When finished, enter exit to return to the VM ssh shell. Your shell prompt should change to confirm you are no longer in the container bash shell.

You were able to connect to a bash shell because the nginx image has a Debian Linux layer which includes bash. Not all images will include bash, but exec can be used to run any supported command in the container.

 12. To list the files in the container's /etc/nginx directory, enter:

docker exec web-server ls /etc/nginx

 This runs the ls command and returns to the ssh shell prompt without using a container shell to execute the command. What commands are supported depends on the layers in the container's image. Up until now you have used two images but don't know how to find more.

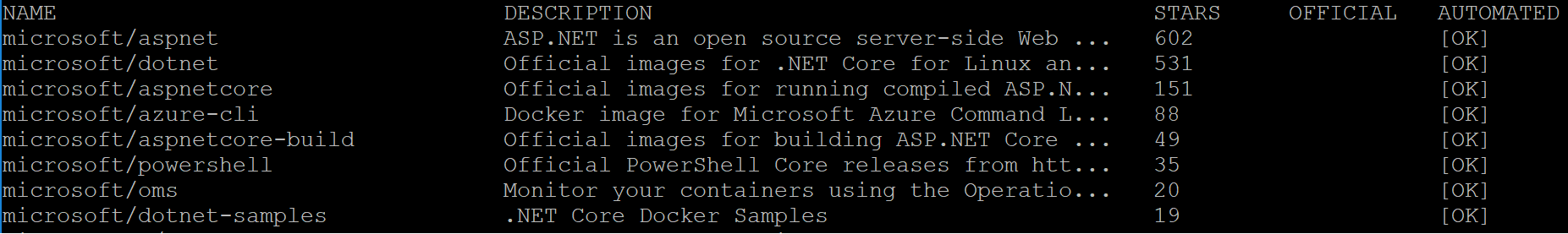
 13. Stop the nginx container:

docker stop web-server

 14. Search for an image that you don't know the exact name of, say an image for Microsoft .NET Core, by entering:

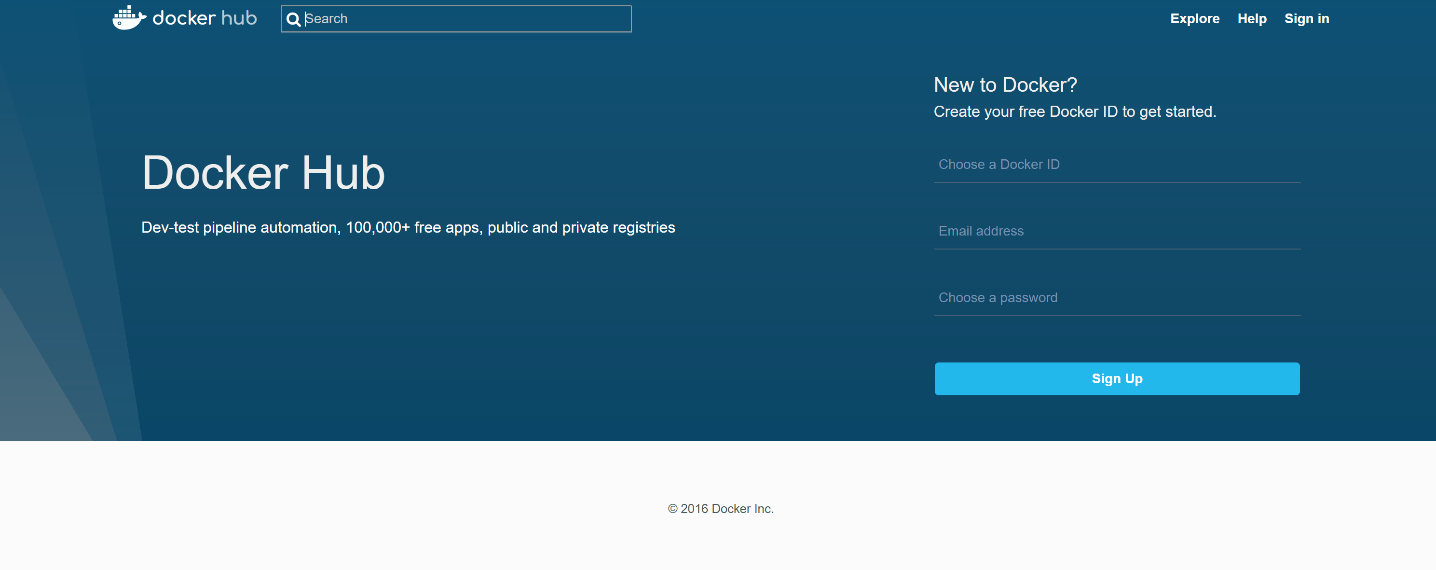
docker search "Microsoft .NET Core"

This searches Docker Hub for images related to the string provided. In this case, the top results related to .NET Core are returned:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid2-f38d9025-78c1-454b-a518-dedbe7c2f033.png)

This can be useful for recalling the name of an image but not very useful for images that have multiple configuration options. For that you should use the web version of Docker Hub.

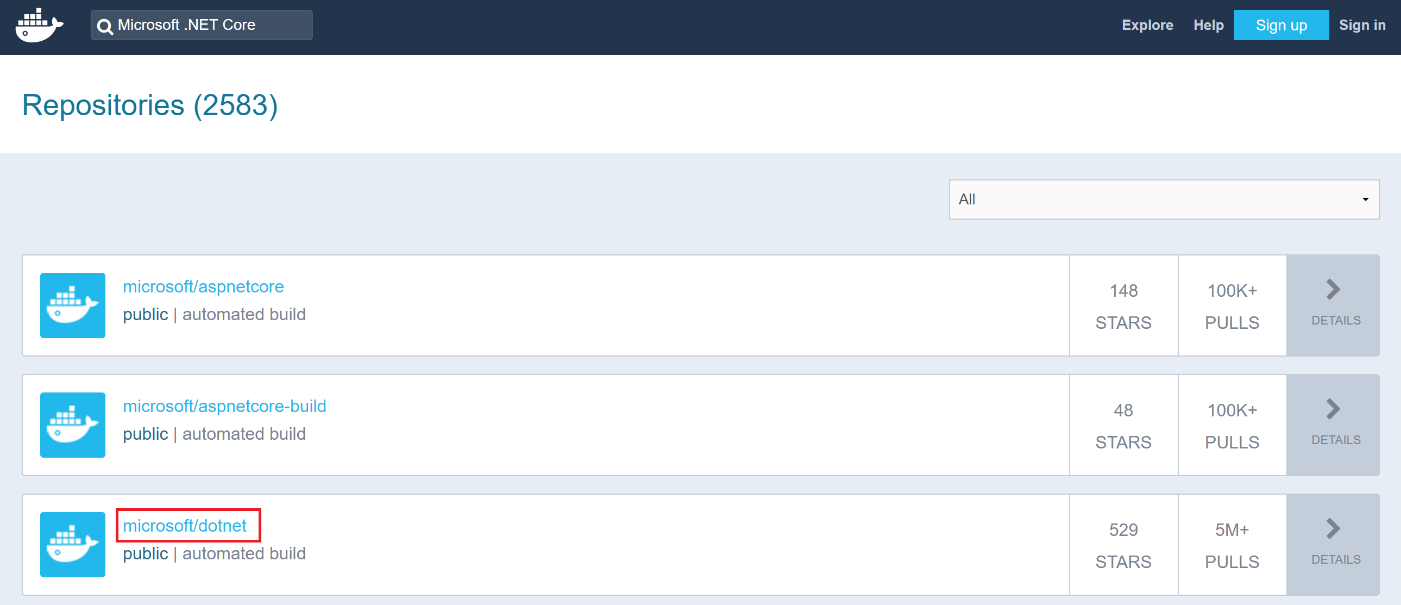
15. Navigate to <https://hub.docker.com>:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid0-859f219f-e5da-4625-b1c3-2e34ddb1d330.png)

This is the Docker Hub web site.

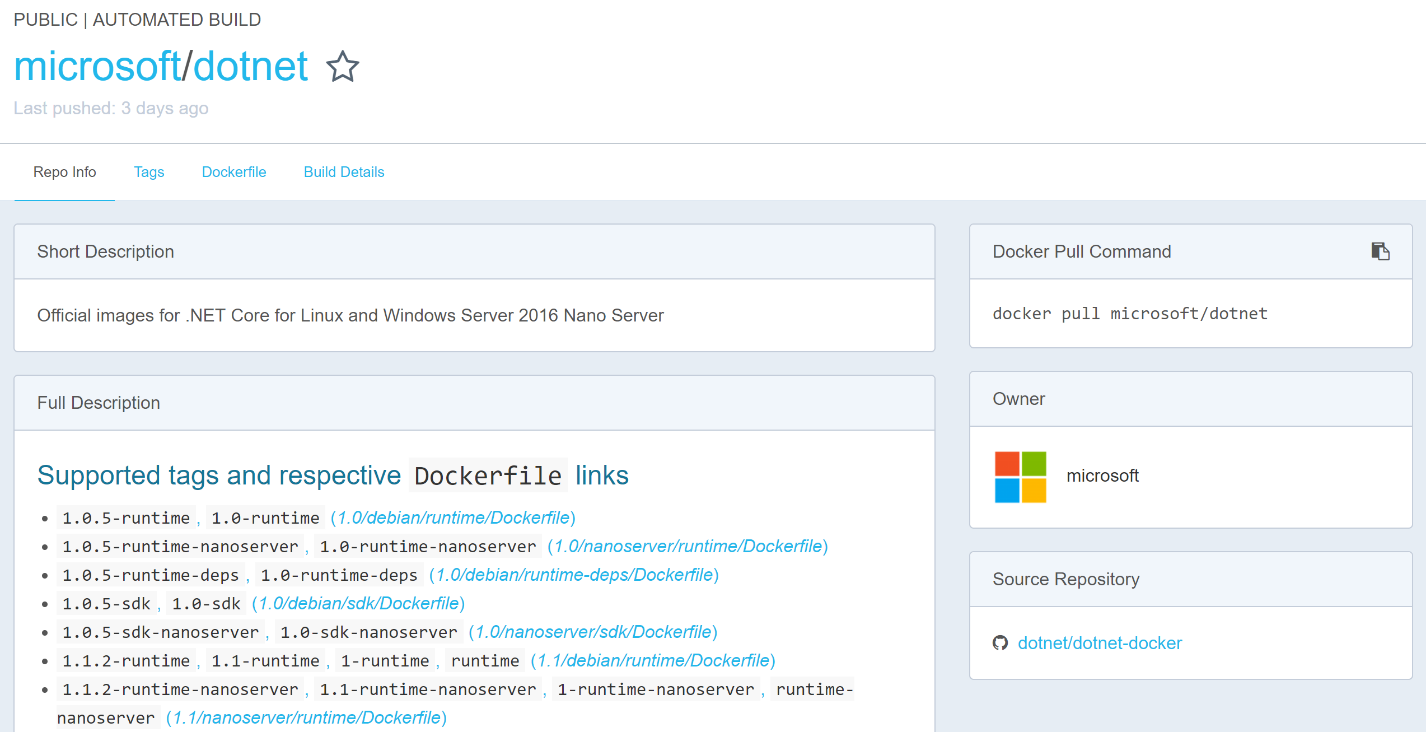
 16. In the search box, enter the same search *Microsoft .NET Core* and press enter.

 17. Click on the **microsoft/dotnet** result

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid4-10e66154-cd16-4b23-b2a1-6f936a9f02be.png)

This is Microsoft's .NET Core repository.

 18. You will be viewing the web page of the microsoft/dotnet repository:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid5-f0ffd1b6-df4d-4779-9f15-7d255aac55ca.png)

You will find all the supported tags along with helpful documentation for using the images. You will usually find useful documentation and examples for images on Docker Hub. You will notice a *Dockerfile* beside each group of tags on the dotnet page. A Dockerfile is a set of instructions for creating an image and it is what you will dive into in the next Lab Step.

**Step 8 Creating Your First Docker Image**

**Introduction**

Docker containers run on top of images. You have seen how to use images on Docker's public registry, the Docker Hub. There are many different images available. It is worth trying to find existing images when you can. Inevitably, you will need to create your own images when making your own applications. In that case, you will still want to invest time into finding the right base layer to add your own layer(s) on top of.

There are a couple of ways to make images. You can use docker commit to create an image from a container's changes. The changes may come from using exec to open a shell in the container like in the previous Lab Step. The other method is using a *Dockerfile*. A Dockerfile is easier to maintain, easier to repeatedly create images from, and distributions easier. You will create a Dockerfile in this Lab Step. Just know that it is possible to create equivalent images using commits.

Dockerfiles specify a sequence of instructions. Instructions can install software, expose network ports, set the default command for running a container using the image, and other tasks. Instructions can really handle anything required to configure the application. Many of the instructions add layers. It is usually a good idea to keep the number of layers to a reasonable number. There is overhead with each layer, and the total number of layers in an image is limited. When the Dockerfile is ready, you can create the image using the docker build command.

You will see how all of this comes together by creating an image of a Python Flask web app that displays “Hello World” . The choice of Python as the example app is arbitrary. You will not focus on the specifics of the programming language. You should be able to repeat the process for any other programming language by following a similar process. Whatever programming language or framework you are working with, you should consult the Docker Hub documentation for the image, as it will usually include advice on how to structure your Dockerfile.

**Instructions**

1. Install Git:

sudo yum -y install git

You will clone a code repository with the Flask app using Git.

 2. Clone the code repository to your virtual machine:

git clone https://bitbucket.org/toorroot/devops\_base.git

 3. Change to the Labs directory:

cd Labs/docker/flask-hello

 4. Create and start editing a Dockerfile using the vi text editor:

vi Dockerfile

*Note:* The name of the Dockerfile must be Dockerfile with an uppercase "D" and all other letters lowercase

 5. Enter the following in the file:

# Python v3 base layer  
FROM python:3  
  
# Set the working directory in the image's file system  
WORKDIR /usr/src/app  
  
# Copy everything in the host working directory to the container's directory  
COPY . .  
  
# Install code dependencies in requirements.txt  
RUN pip install --no-cache-dir -r requirements.txt  
  
# Indicate that the server will be listening on port 5000  
EXPOSE 5000  
  
# Set the default command to run the app  
CMD [ "python", "./src/app.py" ]

The lines beginning with # are comments and explain what each instruction is doing. Make sure you read the comments. Some highlights are:

* FROM sets the base layer image
* COPY . . copies all of the files in the code repository into the container's /usr/src/app directory
* RUN executes a command in a new layer at the top of the image
* EXPOSE only indicates what port the container will be listening on, it doesn't automatically open the port on the container
* CMD sets the default command to run when a container is made from the image

There are more instruction types, but this lab will only focus on those mentioned. After completing the lab, you can review all of the instructions available in Dockerfiles at the [Dockerfile reference web page](https://docs.docker.com/engine/reference/builder/).

 6. Once you have all of the instructions in the Dockerfile, enter :wq to write (save) the file and quit vi.

You are now back at the shell prompt.

 7. Build the image from the Dockerfile:

docker build -t flask-hello:latest .

The -t tells Docker to tag the image with the name flask-hello and tag latest. The . at the end tells Docker to look for a Dockerfile in the current directory. Docker will report what it's doing to build the image. Each instruction has its own step. Steps one and four take longer than the others. Step one needs to pull several layers for the Python 3 base layer image and Step four downloads code dependencies for the Flask web application framework. Notice that each Step ends with a notice that an intermediate container was removed. Because layers are read-only, Docker needs to create a container for each instruction. When the instruction is complete, Docker commits it to a layer in the image and discards the container.

8. Now you can run a container using the image you just built:

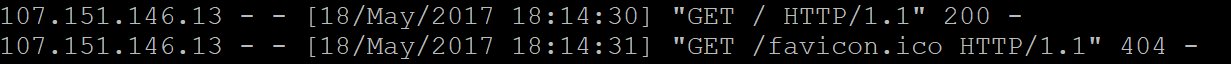
docker run -d --network=host --name flask\_hello flask-hello

This runs a container named advisor and maps the container's port 5000 to the host's port 80 (http). This time you include -d to run in detached mode.  That is why you don’t see output and you don't have the shell prompt returned to you. If you did run with -d, you could get the same information from docker logs.

 9. Request http service for 5000 port:

curl http://127.0.0.1:5000

10. Return to the shell and notice that some web requests will have been logged corresponding to your browser requests:

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid2-f12d360d-8ed4-4531-a1f5-f3f8a15e1b35.png)

There are two requests because the browser automatically requests a favicon in addition to the page content.

 11. Terminate the running container.

docker stop flask-hello

You are now are at the shell prompt.

## Step 9 Cleaning Up Your Docker Containers and Images

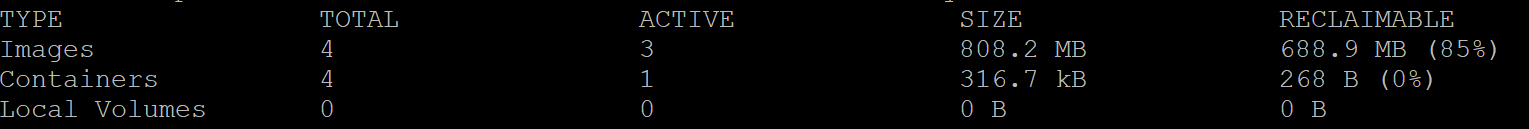
### Introduction

As you use Docker more and more, you will accumulate more and more image layers and containers. Images can exceed gigabytes in size. Stopped containers tend not to use a significant amount of disk space, but can pollute the output from some Docker commands. You will learn how to reclaim the space from unneeded images and clear out needed space in this Lab Step.

### Instructions

1. Get an understanding of how much space is being used with the following system management command:

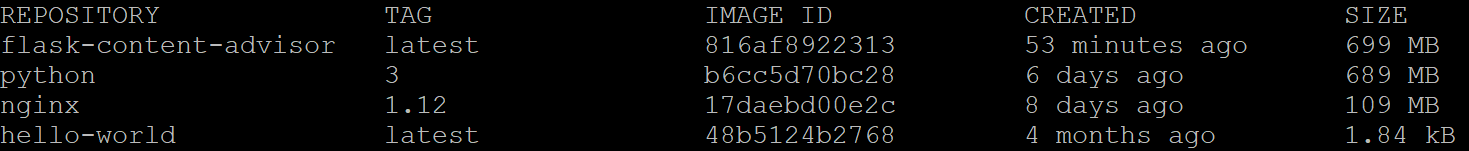
docker system df

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid0-8612ab01-064e-4238-9cd2-568d334bf2ff.png)

This breaks down the disk usage into images and containers. Images are taking up the majority of the space.

 2. To see which images are taking up the most space, enter:

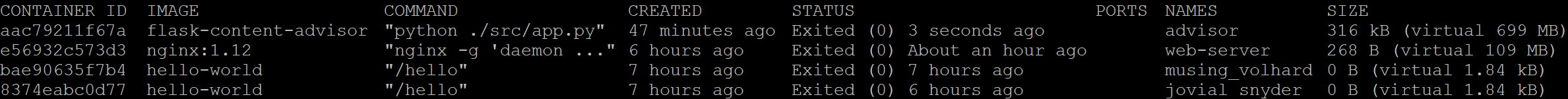
docker images

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid1-255e9929-1bcd-48a5-a286-aa1b35cec4f9.png)

The **flask-hello** and **python** images are both around 700MB. That's because the flask-hello image is built on top of the python image. The difference between the two is how much additional space was added when you built the image using a Dockerfile. Notice that the output from df is smart enough to not double count the layers since they are shared between the images.

3. You can get more information on the size of containers with the following command:

docker ps -a -s

[](https://assets.cloudacademy.com/bakery/media/uploads/blobid0-4ce40095-9318-4b96-bf22-8e6aa4a81583.png)

The -s includes an extra column at the end showing the size of the container layer and the virtual size. The virtual size includes the image size and the container layer size.

4. To remove the stopped web-server container, enter:

docker rm web-server

The container must be stopped for you to be able to remove it. Confirm it has been removed with docker ps -a.

5. The nginx image can now be removed because there are no containers depending upon it:

docker rmi nginx:1.12

Each layer in the image is deleted. If there were image layers shared with other images that were used by containers, Docker would be smart enough to only remove the layers not being used. You will observe this now.

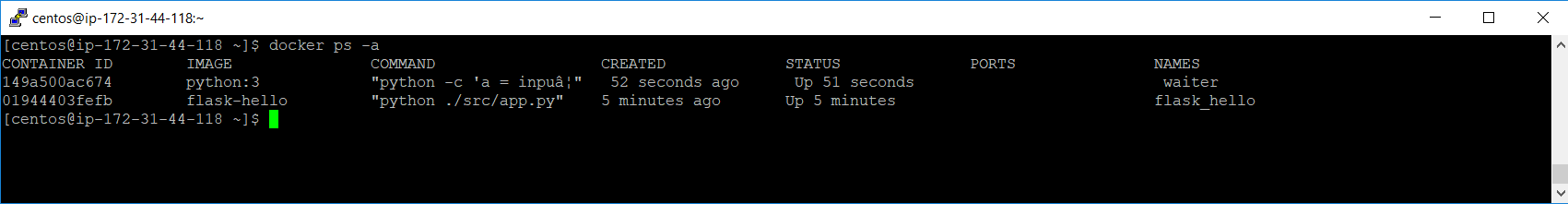
6. Run a container based on the Python 3 image:

docker run --name waiter -d -t python:3 python -c 'a = input("a")'

This will run a container that will endlessly wait for someone to input the value of a variable. This is an example of not using the default command of an image. The command in this case is everything after the image: python -c 'a = input("a")'. The Python 3 image is the base layer of the flask-hello image you built.

 7. List all the containers:

docker ps -a



Notice the **waiter** container has a **STATUS** of **Up**, meaning it is running.

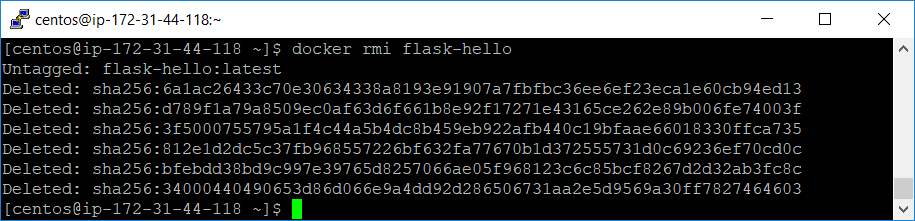
8. Remove the advisor container:

docker rm flask-hello

Now you have one container using the Python 3 image, but none using the flask-hello image.

9. Remove the flask-hello image:

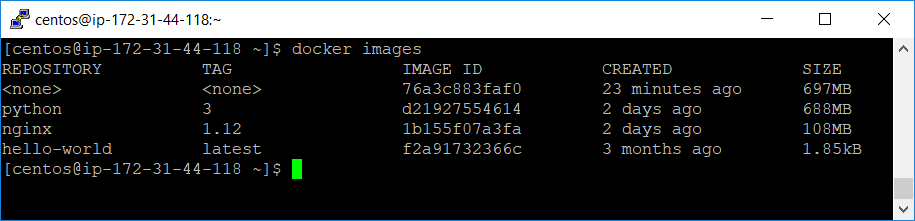
docker rmi flask-hello



This displays a delete output for each layer that is no longer being used. There are 8 layers deleted in total.

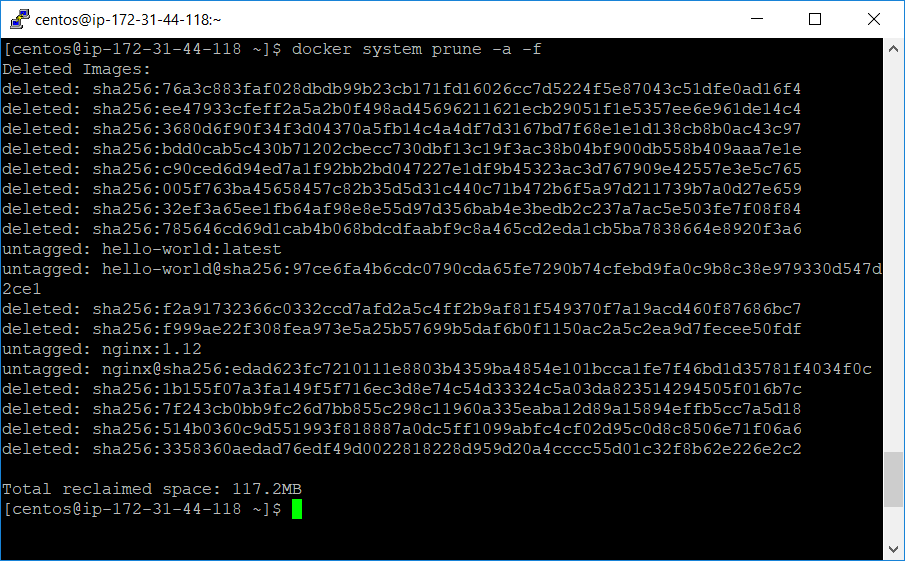
10. Verify that the Python 3 image wasn't touched by the operation:

docker images



11. To clean up everything not running, enter:

docker system prune -a -f



This prunes all containers and images that aren't used by running containers. This is a convenient way to clean up if you haven't been regularly removing unused images and containers. Verify that only the one running Python 3 container and its image are left with docker system df.