



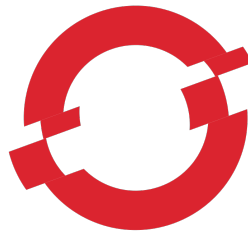
redhat®

THE NEW PaaS:

Using Docker and Containers to Simplify Your
Life and Accelerate Development on AWS

LAB 1: DOCKER

Version 1.2



OPENSIFT



Copyright © 2014 Red Hat, Inc.

Red Hat, the Shadowman logo, and the OpenShift logo are trademarks of Red Hat, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

Table of Contents

Introduction	4
Overview	4
What is Docker?	4
https://github.com/docker/docker	4
Technical Knowledge Prerequisites	4
Topics Covered	4
Sign in to the AWS Management Console	4
Using qwikLABS™ to sign in to the AWS Management Console	4
Setting Up SSH Access for Labs 1, 2, and 3	6
To set up SSH on Linux, OS X, and Windows with Cygwin	7
To set up SSH on Windows using PuTTY	7
Module 1: Docker Basics	9
1. Search Docker images from the Docker Hub	9
2. Pull a Docker Image from the Docker Hub	10
3. List Downloaded Images	11
4. Running a Container	11
5. Running a container as a “daemon”	12
Module 2: Modifying Docker Images	14
1. Modifying a Docker image using a Docker file	14
2. Modifying a docker image by making changes in a container	16
Module 3: Inter-Container Communication	19
1. Starting a Postgresql Docker Image	19
2. Starting the spousty/myfedora Image to talk to the DB Container	19
3. Testing the Connection	21
Conclusion	22
Additional Resources	22

Introduction

Overview

In this lab we will explore the basic operations of a docker host. First we will learn basic docker image operations, then we will launch and interact with terminating and non-terminating docker containers. Finally, we will learn how to build and publish our own docker images.

What is Docker?

Docker an open platform for developers and sysadmins to build, ship, and run distributed applications. You can learn more here:

<https://github.com/docker/docker>

Technical Knowledge Prerequisites

To successfully complete this lab, you should be familiar with SSH or PuTTY, and with the Linux command-line environment.

Topics Covered

This lab will take you through docker operations, including:

- Searching and downloading docker images
- Running interactive and daemonized docker containers
- Building and publishing new docker images

Sign in to the AWS Management Console

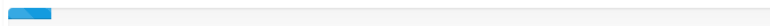
Using qwikLABS™ to sign in to the AWS Management Console

Welcome to this self-paced lab! The first step is for you to sign in to Amazon Web Services.

1. To the right of the lab title, click **Start Lab**. If you are prompted for a token, use the one you received or purchased.

Note: A status bar shows the progress of the lab environment creation process. The AWS Management Console is accessible during lab resource creation, but your AWS resources may not be fully available until the process is complete.

** Create in progress...*

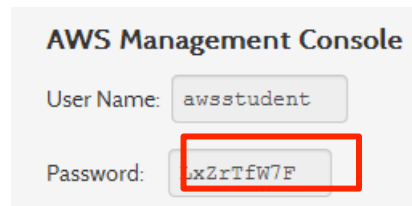


2. On the lab details page, notice the lab properties.
 - a. **Duration** - The time the lab will run before automatically shutting down.
 - b. **Setup Time** - The estimated time to set up the lab environment.
 - c. **AWS Region** - The AWS Region in which the lab resources are created.

Duration (minutes): 600
Setup Time (minutes): 0
AWS Region: [us-east-1] US East (N. Virginia)

Note: The AWS Region for your lab will differ depending on your location and the lab setup.

3. In the AWS Management Console section of the qwikLAB™ page, copy the Password to the clipboard.



AWS Management Console

User Name:

Password:

4. Click the Open Console button.



5. Log into the AWS Management Console using the following steps.
 - a. In the **User Name** field type **awsstudent**.
 - b. In the **Password** field, paste the password copied from the lab details page.
 - c. Click **Sign in using our secure server**.

Amazon Web Services Sign In

Please enter the AWS Identity & Access Management (IAM) User name and password assigned by your system administrator to sign in.

AWS Account: 83280962232



User Name:

Password:

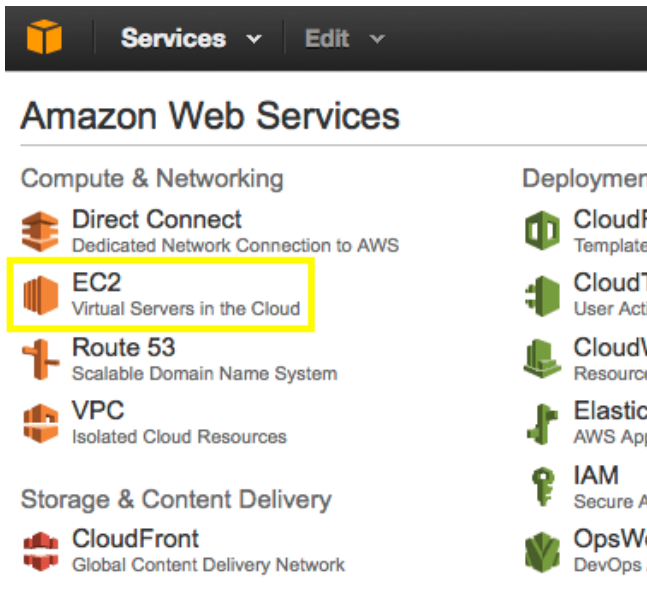
Please contact your system administrator if you have forgotten your user credentials.

[Sign in using AWS Account credentials](#)

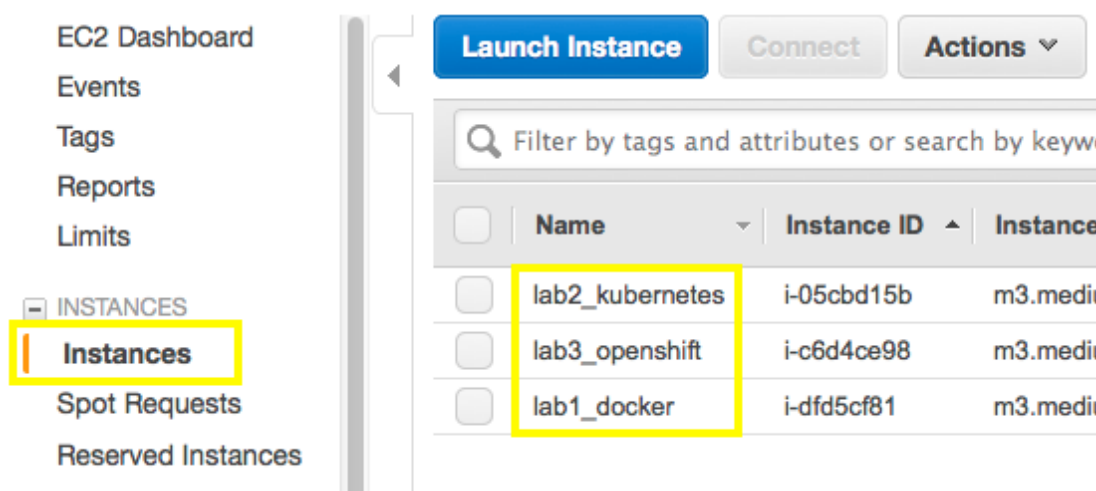
Note: The AWS account is automatically generated by qwikLAB™. Also, the login credentials for the awsstudent account are provisioned by qwikLAB™ using AWS Identity Access Management.

Setting Up SSH Access for Labs 1, 2, and 3

Once you have logged in to your temporary AWS account, select the EC2 item from the control panel:



Once there, you will see three running AWS instances:



Each instance is named for the lab that is associated with that instance. We will be working with each instance separately as we tackle each lab.

If you select the `lab1_docker` instance in the list, you will see information about the instance in the bottom portion of the browser window. This information includes the instance's

public IP address:

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS
lab2_kubernetes	i-05cbd15b	m3.medium	us-west-1a	running	2/2 checks ...	None	ec2-54-215-168-1
lab3_openshift	i-c6d4ce98	m3.medium	us-west-1a	running	2/2 checks ...	None	ec2-54-183-251-3
lab1_docker	i-dfd5cf81	m3.medium	us-west-1a	running	2/2 checks ...	None	ec2-54-193-61-91

Instance: i-dfd5cf81 (lab1_docker)		Public DNS: ec2-54-193-61-91.us-west-1.compute.amazonaws.com		
Description	Status Checks	Monitoring	Tags	
Instance ID	i-dfd5cf81		Public DNS	ec2-54-193-61-91.us-west-1.compute.amazonaws.com
Instance state	running		Public IP	54.193.61.91
Instance type	m3.medium		Elastic IP	-

For each lab, you will need to look up the appropriate lab instance's public IP address from the EC control panel.

To set up SSH on Linux, OS X, and Windows with Cygwin

Follow these steps to connect with your hosts via SSH:

1. First, download the private key that is associated with these instances. Using curl, the command is:

```
curl -O -J http://reinvent-hripps.rhcloud.com/aws-reinvent-2014.pem
```

2. Change the permissions on the .pem file so that SSH will not throw an error:

```
$ chmod 600 /path/to/aws-reinvent-2014.pem
```

Once this has been done, you can connect with any of your instance by running this ssh command with the relevant public IP address from the EC2 control panel:

```
ssh root@<public_ip_address> -i /path/to/aws-reinvent-2014.pem
```

Note that in some of the labs, you will need to open multiple SSH sessions against the same instance. You can do this by opening multiple shells and running the SSH command in each one.

To set up SSH on Windows using PuTTY

Using PuTTY, you can connect to your lab instances by following these steps:

1. Download the puTTY-compatible private key that is associated with these EC2 instances. Using a web browser, navigate to:

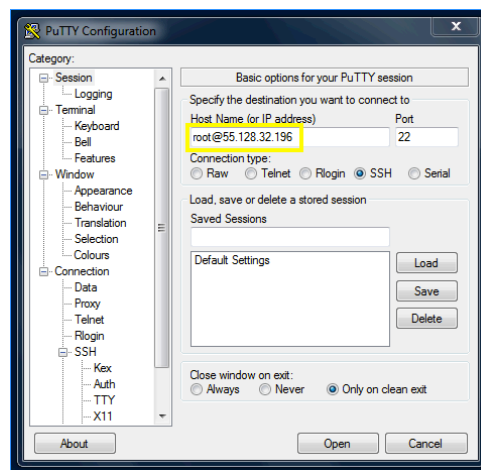
```
http://reinvent-hripps.rhcloud.com/aws-reinvent-2014.ppk
```

Save the file to your local system, taking care to *save it as a plain text file*, and take note of where you saved it. You may need to change the extension back to `.ppk` if the browser automatically added `.txt`.

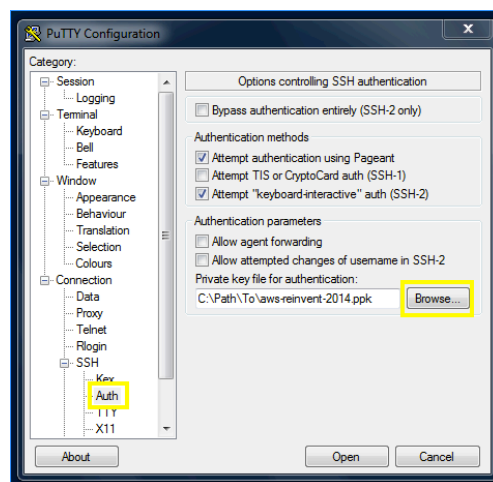
2. Start PuTTY. In the default view, look for the Host Name (or IP Address) field. Change the value here to:

```
root@<public_ip_address>
```

Where the public IP address that you use is the relevant public IP addresses from the EC2 console. (Make sure you pick the instance that corresponds to the lab that you are currently working on.)



3. Now, using the Category field on the left-hand side of the PuTTY window, navigate to Connection => SSH => Auth:



Next to the Private key for authentication: field, press the Browse button and select

the *.ppk file that you downloaded in step 1.

4. Finally, press the Open button to connect to the host. You will see a security alert the first time you connect to a host; press Yes to accept the host and continue connecting.

In some of the labs, you will need to open multiple SSH sessions against the same instance. You may want to take advantage of PuTTY's ability to save and load SSH profiles to simplify the task of opening the additional sessions.

Module 1: Docker Basics

In this module, we will exercise all of the basic Docker functionality to search, retrieve and use Docker images.

1. Search Docker images from the Docker Hub

Docker images can be searched in much the same way you search any collection. Run the following command to learn about one of the images that we will be using in this lab:

```
$ docker search fedora/apache
```

You should see something like this:

```
$ docker search fedora/apache
NAME                DESCRIPTION          STARS   OFFICIAL   AUTOMATED
fedora/apache              26           [OK]
fedora/systemd-apache    5             [OK]
...
```

The first part of the name is the namespace (in this case Fedora project) and the second part is the package, which usually is descriptive of the OS or the image's purpose. As for the other fields:

- Stars: Docker images can be rated by users, the more stars a container has, the more popular it is.
- Official: The folks at Docker maintain a number of "official" images, as denoted by this field.
- Automated: Docker supports automated builds; any images that are produced by the automated build process are flagged with this.

EXERCISE

Go ahead and search for `fedora` with more than 5 stars and see what you get back

2. Pull a Docker Image from the Docker Hub

Recall from the presentation that Docker images are actually comprised of layered file systems. When you perform a docker pull, you initiate the parallel download of all of the file system components that make up your requested Docker image. Try it now by running the following:

```
$ docker pull fedora/apache:latest
```

When it is done, you should see output similar to this:

```
$ docker pull fedora/apache:latest
Pulling repository fedora/apache
2e11d8fd18b3: Download complete
511136ea3c5a: Download complete
ff75b0852d47: Download complete
0dae8c30a0b2: Download complete
84f33df93401: Download complete
24b116bb2956: Download complete
a7f290a6f21d: Download complete
eb86e2be11d4: Download complete
c06d2cba0d4a: Download complete
f0b140ef8cdd: Download complete
b05601b61180: Download complete
```

In the case of this particular Docker image, each of these filesystems represents a different tagged version of the image. Most Docker images are composed of multiple filesystem layers.

EXERCISE

Run the following command to see what happened with each layer

```
docker history fedora/apache
```

3. List Downloaded Images

This command shows you all of your locally available Docker images:

```
$ docker images
```

You should see something like this:

```
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          VIRTUAL SIZE
fedora/apache       latest      2e11d8fd18b3     9 days ago      554.1 MB
....
```

Since we pre-installed other Docker images on the EC2 machine you will see all the other images on your machine as well.

4. Running a Container

To start this exercise, first familiarize yourself with `docker ps`:

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

Nothing is running right now, so the list is empty. Now kick off a simple container with `docker run`:

```
$ docker run -i -t fedora/apache /bin/echo 'hello world'
hello world
```

Here's a breakdown of what just happened:

- `-i`: Make an interactive connection to the container by grabbing STDIN
- `-t`: Assign a pseudo-terminal in the container
- `fedora/apache`: the image to run as a container
- `/bin/echo 'hello world'`: A command passed to the container's pseudo terminal via STDIN

Notice that we still haven't explicitly identified which tagged version of the image we want to run. In this situation, Docker defaults to using the version tagged with `latest`.

Now run `docker ps` again:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS        NAMES
```

Still nothing! This is because the Docker container that we ran isn't running anymore. It stopped as soon as its primary process exited. Try running `docker ps` with the `-l` argument to see the result of the last container invocation:

```
$ docker ps -l
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS        NAMES
5e05cd28f29c   fedora/apache:latest   /bin/echo 'hello wor    About a
minute ago    Exited (0) About a minute ago
silly_kowalevski
```

Notice that docker assigned a random id and name to our container when it spun it up. We will revisit the name piece later in the exercises.

EXERCISE

Now make your docker image run `top`. Remember, to quit `top` just type `q`.

5. Running a container as a “daemon”

While having an image spin up and run command is “neat”, now it's time to do some more useful tasks. Let's get our image to start up and serve HTML content through Apache Web Server.

Enter the following command:

```
$ docker run -d -P fedora/apache
798a3749cd7e0721a59af2879ecfebfd1096cb1c925ab73549102c06b788a4e5
```

Here is a breakdown of the flags:

- `-d` means run the container in the background (which means don't run and exit)
- `-P` means map any required network ports inside our container to ports on the host (basically automatically do port-mapping)

If you remember back to when you did the `docker history` command, one of the changes was to have the image spin up Apache when it is spun up as a container. The end result is that by running the command above we now have Apache running and serving up `index.html`.

Now let's find out what port we should use to get to Apache. Time to use the trusty `docker ps` command:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	PORTS
798a3749cd7e	fedora/apache:latest	/run-apache.sh	5
seconds ago	Up 5 seconds	0.0.0.0:49153->80/tcp	
elegant_pare			

`docker ps` shows something even without the `-l` because we used the `-d` flag when calling `docker run`. What is also different is you can see that there is an entry for ports:

```
0.0.0.0:49153->80/tcp
```

This line indicates that port 49153 on the host machine points to port 80 on the container. It also is only exposing TCP traffic. Based on this we can execute the following command on our host machine:

```
$ curl -v 0.0.0.0:49153
* About to connect() to 0.0.0.0 port 49153 (#0)
*   Trying 0.0.0.0... connected
* Connected to 0.0.0.0 (0.0.0.0) port 49153 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu)
libcurl/7.19.7 NSS/3.15.3 zlib/1.2.3 libidn/1.18 libssh2/1.4.2
> Host: 0.0.0.0:49153
> Accept: */*
```

```
>
< HTTP/1.1 200 OK
< Date: Thu, 09 Oct 2014 18:35:13 GMT
< Server: Apache/2.4.10 (Fedora)
< Last-Modified: Mon, 29 Sep 2014 12:28:50 GMT
< ETag: "7-5043368b3a480"
< Accept-Ranges: bytes
< Content-Length: 7
< Content-Type: text/html; charset=UTF-8
<
Apache
* Connection #0 to host 0.0.0.0 left intact
* Closing connection #0
```

Voila, you have successfully spun up a Docker image and have it serving content. If you want to stop this Docker image from running just enter:

```
$ docker kill elegant_pare
```

You could have also used the container ID in place of the name.

Time for us to start modifying Docker images.

Module 2: Modifying Docker Images

In this module we are going to learn how to modify, save, and use the new Docker images. To start with we are going to modify a Docker file to add some software to our image. After that we will make a change to a container and save that as a new image. By the end of this module you should be comfortable modifying Docker images for your own use.

1. Modifying a Docker image using a Docker file

One of the author's favorite text editors is `nano`, since it is simple and easy to understand. Unfortunately it is not installed on the `fedora/apache` Docker image. To check for yourself try running the following command:

```
$ docker run -i -t fedora/apache /bin/nano
2014/10/09 21:52:24 Error response from daemon: Cannot start
container
```

```
28f59567a655f898ad2782e328b8d9507a1320517b75b2626241829d1c3ef047:  
exec: "/bin/nano": stat /bin/nano: no such file or directory
```

The output of the command is Docker's way of indicating that a binary is not installed. To remedy this we are going to create a new image with `nano` and the `postgresql` client, which we need for a later module.

First you need to make a new Docker file. Execute the following commands in the shell:

```
$ mkdir nano-pg  
$ cd nano-pg  
$ touch Dockerfile  
$ nano Dockerfile
```

We are editing with `nano` for the reasons stated above but feel free to use `Vi` or `Emacs`. Inside this file we are going to specify what we want Docker to build when making our image. Please enter the following into the text file :

```
# This is a comment  
  
#pick the image we used before as our starting place  
FROM fedora/apache:latest  
  
#Identify yourself as the maintainer. Please change to your name  
and email  
MAINTAINER Steven Pousty <spousty@redhat.com>  
  
#Run yum to install our two packages  
RUN yum install -y nano postgresql
```

If you used `nano` then please hit `ctrl-x, y`, and then hit `<Enter>`. What you just did was create a file that specifies how Docker will build a new image. First you told Docker to use the `fedora/apache` image as the starting point. Then, you specified that you are the maintainer of this new image. Finally you tell Docker to run the command to install `nano` and `postgresql` without any user intervention.

Now we just need to have Docker use this build file. At the command prompt enter the following command (please substitute your own namespace if you want):

```
$ docker build -t "spousty/myfedora:1.0" .
```

We are telling Docker to build a new image tagged with my namespace, a new name, and tagged with 1.0. Since we are putting the final tag on the name, and it is not “latest” or blank, this will require us to use this tag whenever we refer to the image.

Now we you do:

```
$ docker images
```

You should see your image in the list. And when you do this command:

```
$ docker run -i -t spousty/myfedora:1.0 /bin/nano
```

You will get nano running in the container. In your terminal – WINNING!

EXERCISE:

Go ahead and make another Dockerfile in a new directory. This time install another package like python and then test to see if it is installed.

2. Modifying a docker image by making changes in a container

Another way to make changes to an existing image is modify fiels in the container and then save the modified file system as a new image. For this exercise we are going to update the index.html that Apache is serving up by default.

If you look back at the docker history on fedora/apache you can see a FS layer that looks like this:

```
eb86e2be11d4 11 days ago /bin/sh -c echo "Apache" >> /var/www/html/ind  
7 B
```

This tells us that the index page for Apache is coming from /var/www/html/index.html. To edit that page we need to make sure we start the container and leave it running after we are done editing. First start the container to give us a command prompt:

```
docker run -i -t spousty/myfedora:1.0 /bin/bash
```




Now you need to open your editor to edit the index.html. Here is how we did it with nano:

```
$ nano /var/www/html/index.html
```

Please remove the word “Apache” from the file and put in whatever text you would like to see. I changed mine to:

```
Sudo make me an image
```

Now we have changed the file and you can exit the container by typing `exit`. Before doing anything else enter:

```
$ docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
4a940d5590f6	spousty/myfedora:1.0	/bin/bash	49 seconds ago	Exited

You need the container id when you want to commit the changes. To commit your changes do the following command:

```
$ docker commit -m="updated index.html" -a="Steve Pousty" -run '{"Cmd":  
["/run-apache.sh"]}' 49fca79a5e67 spousty/myfedora  
f957afdba618431fb3dc9e8cac6b5e81ad5c4d671a8fd1418bfd55865cf61b6
```

This command commits changes from the container to a new image. We need to add the `-run` flag because by default, the commit will set the default action to be the action that was last run in the container. In our case, that was `/bin/bash`, which means it doesn't really run as a daemon anymore. The `-run` sets what the default command is to run on the image.

The `-m` and `-a` allow us to set the metadata on the image like we did in our Docker file. The unique id is for the container (you could also use the container's name) and then finally we give the new image we want. By leaving off the final part of the tag, Docker will make it `:latest`.

Now let's run this image again like we did before, but please be aware that the port number will change.

```
$ docker run -d -P spousty/myfedora  
52af4105cfbbf306ecb608b6170c59332e881146696df5af804c7e5d482d72fc
```



```
$ docker ps

CONTAINER ID          IMAGE                COMMAND
CREATED              STATUS              PORTS
NAMES

52af4105cfbb         spousty/myfedora:latest  /run-apache.sh
3 seconds ago       Up 2 seconds        0.0.0.0:49159->80/tcp
hopeful_kowalevski

$ curl 0.0.0.0:49159

Sudo make me a sandwich
```

Success! This would be the pattern you have to follow to make changes to your website.

EXERCISE:

Stop the container, modify the file to say something else like “Look ma I edited HTML”, save the image again maybe with a different tag (or not). Finally, see if you can get the changes to stick so that you can start up the image and you get the new text when you curl the URL for the container.

You have just learned the two main ways to modify and commit your Docker images. A good topic to follow up on this might be how to publish your Docker image to Dockerhub for consumption by others. In this workshop we are not going to look at how to enable communication between two Docker containers.

Module 3: Inter-Container Communication

In most of your current architectures you probably separate your Web Server from your Database (virtual) server. This section will show how to build that type of architecture with Docker containers. Of course, having Docker containers communicate has more uses than a web server and a database, but the pattern you would use stays the same.

1. Starting a Postgresql Docker Image

When you run `docker images` you can see that we have already an image titled `training/postgres`. This image was created by Docker for their tutorials and is based off of Ubuntu with Postgresql 9.3. We thought this would be a good way to show you that you can run many different linux containers on the same underlying kernel.

When we start the container we are going to take advantage of Docker's ability to give our containers meaningful names. Start the postgresql image as a container and name it `db` like so:

```
$ docker run -d -P --name db training/postgres
```

By using the `-name` flag we can assign a name to our container rather than the default behavior where a random name is assigned to the container. We can now use `db` as the container name rather than remembering the random digit sequence or the random name. In general, when doing real work, we recommend you name your images so your architecture is easier to follow.

Let's check which ports our image is exposing:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	PORTS	NAMES
ba9b681f21b3	training/postgres:latest	su postgres -c '/usr	0.0.0.0:49160->5432/tcp	6 db
seconds ago	Up 4 seconds			

You can see that Docker has mapped port 49160 to the port 5432 in the container, which is the default port for Postgresql. And now let's link it to our new image we created.

2. Starting the spousty/myfedora Image to talk to the DB Container

First make sure your new image is not running in a container by doing a `docker ps`. If it is go ahead and kill the container and actually remove the container.



```
$ docker kill hopeful_kowalevski  
hopeful_kowalevski  
$ docker rm -f hopeful_kowalevski  
hopeful_kowalevski
```

Now we are going to start up that image again but this time we are going to link it to our PostgreSQL container. We are also going to name the container this time to make it easier to remember than `hopeful_kowalevski`. Finally, we will also just start at the command prompt so we can use the PostgreSQL command line client. Actually hooking up the web piece to the database is beyond the scope of this class.

```
sudo docker run -i -t --name web --link db:postdb  
spousty/myfedora /bin/bash
```

The link part of this command is of the form

```
--link name:alias
```

Where `name` is the name of the container you want to link to and `alias` is for the link inside the container you are about to spin up.

Once you are at the command prompt let's go ahead and take a look at what the `--link` option did inside this container. First you will see that Docker added some environment variables to this container. To see the environment variables, execute the following command:

```
$ env |grep POSTDB  
POSTDB_NAME=/web/postdb  
POSTDB_PORT=tcp://172.17.0.31:5432  
POSTDB_PORT_5432_TCP_ADDR=172.17.0.31  
POSTDB_ENV_PG_VERSION=9.3  
POSTDB_PORT_5432_TCP_PORT=5432  
POSTDB_PORT_5432_TCP=tcp://172.17.0.31:5432  
POSTDB_PORT_5432_TCP_PROTO=tcp
```

So you can see that our web container now has assigned mappings for this container to attach to the db container. If we look in `/etc/hosts`, we will see that Docker also gave an ip mapping for the container aliased `postdb`:

```
bash-4.2# cat /etc/hosts |grep postdb
```

```
172.17.0.31 postdb
```

All the “wiring” is in place, it is time to see if this all really works.

3. Testing the Connection

Here comes the easy part! You will need one piece of background information; Docker made the username and password for PostgreSQL equal to `docker:docker`. And with that let's try to execute some SQL commands in the `postgres` container from the web container. Enter the following commands at the terminal in the web container:

```
$ psql -h postdb -p $POSTDB_PORT_5432_TCP_PORT -V
psql (PostgreSQL) 9.3.5
```

Now let's just log in and see what databases there are in the database

```
$ psql -h postdb -p $POSTDB_PORT_5432_TCP_PORT -U docker
psql (9.3.5, server 9.3.4)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.
```

```
docker=# \db
      List of tablespaces
  Name      | Owner   | Location
-----+-----+-----
 pg_default | postgres |
 pg_global  | postgres |
(2 rows)
```

To exit out of PostgreSQL just type `\q <enter>`. And with that you know enough to proceed on to the other labs.

EXERCISE:

In the command above please replace `-h db` with `-h` and the proper environment variable to connect to the container.



Conclusion

Congratulations! You now have successfully:

- Worked with Docker images
- Started and stopped Docker Containers
- Made your own Docker images with a Docker file
- Made your own Docker images by making changes in the container
- Enable communication between Docker containers

NOTE: If you will be continuing on to **Lab 2: Kubernetes**, then *do not log out* of your AWS Console!

Additional Resources

For more information about Docker, Kubernetes and OpenShift, go to the [OpenShift Origin repo on GitHub](#).