Reference: <https://github.com/microsoft-dx/DockerLab>

# Docker Hands-on Lab

This is a hands-on lab for getting started with [Docker](https://www.docker.com/) using [Microsoft Azure](https://azure.microsoft.com/).

# Setup - Create a Free Azure Subscription and set up Ubuntu Docker Host

In order to create a free Azure Subscription, you can either use

* the free monthly Azure credit which comes with the (also free) [Visual Studio Dev Essentials](https://azure.microsoft.com/en-gb/pricing/member-offers/vs-dev-essentials/) offer.
* a [Microsoft Azure Pass](http://www.microsoftazurepass.com/) - provided you have received a code from Microsoft representatives or partners **[If you are an attendee at OSI Days, India 2016, you will receive this]**
* a standard [Azure free trial](https://azure.microsoft.com/en-us/free/).

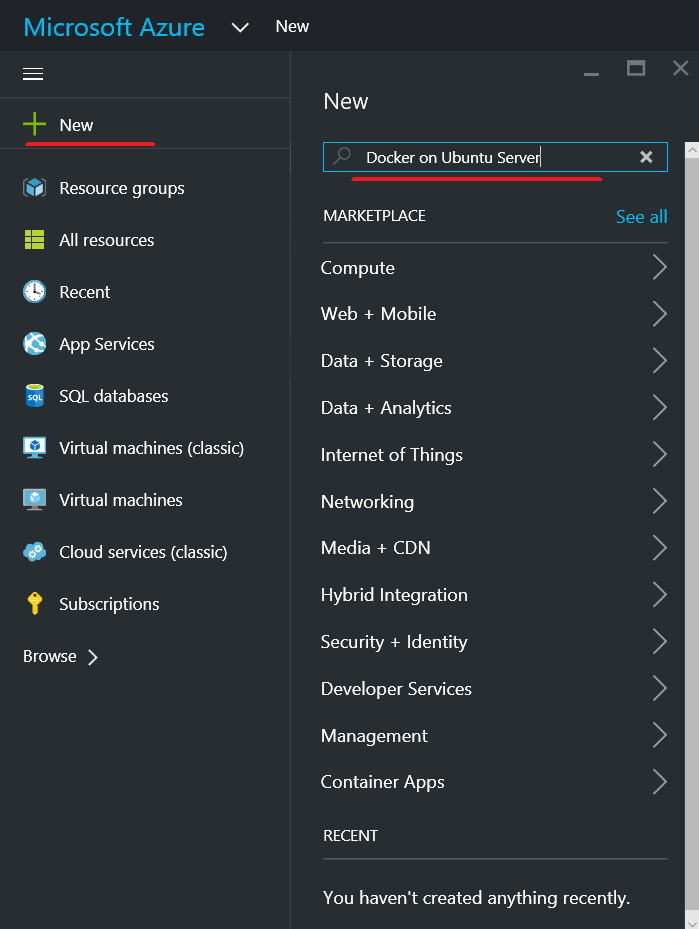
## Setup Docker Host

There are multiple ways of provisioning a Linux-based Docker host.

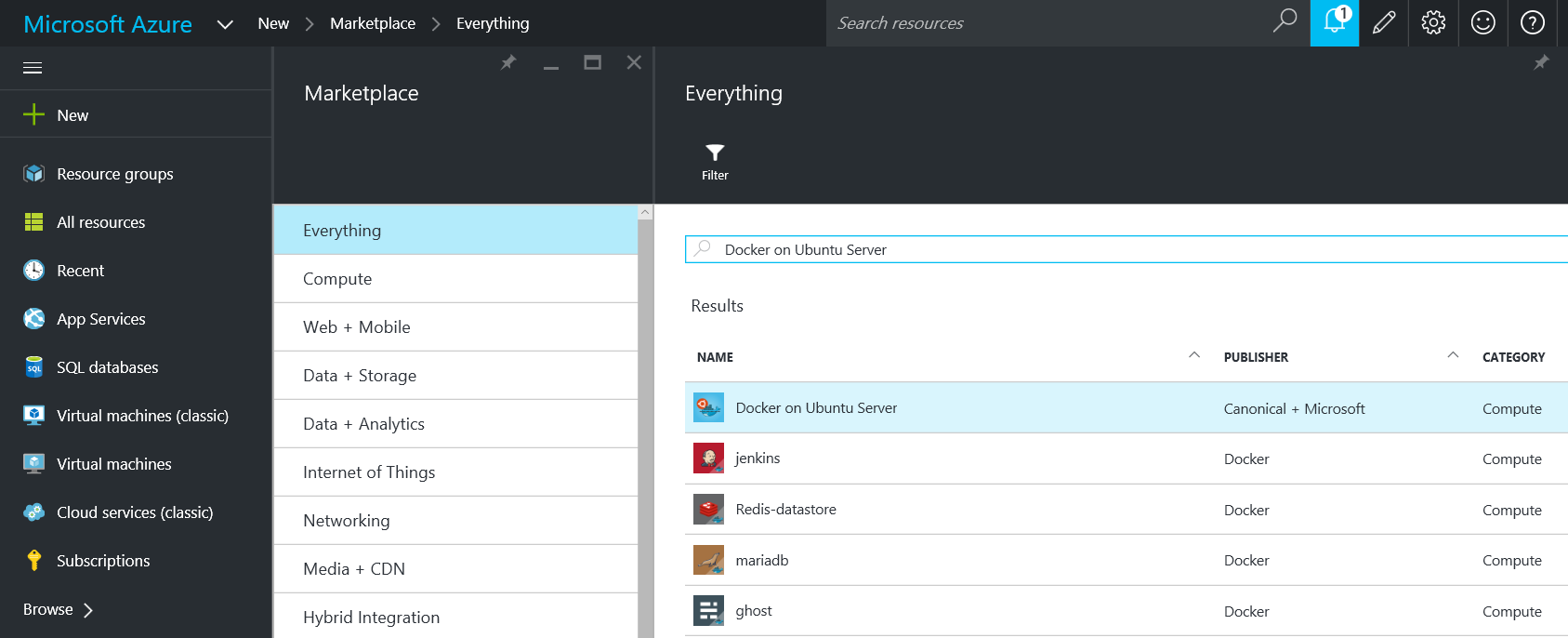
1. Install a [Linux OS on a virtual machine and install Docker](http://docs.docker.com/engine/installation/ubuntulinux/)
2. Use a [pre-existing Linux image from Azure and install Docker](https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-docker-with-portal/)
3. Spin-up an already existing Docker image from the Azure Marketplace

For this lab we will use option 3 - the Ubuntu + Docker image from the Azure Marketplace.

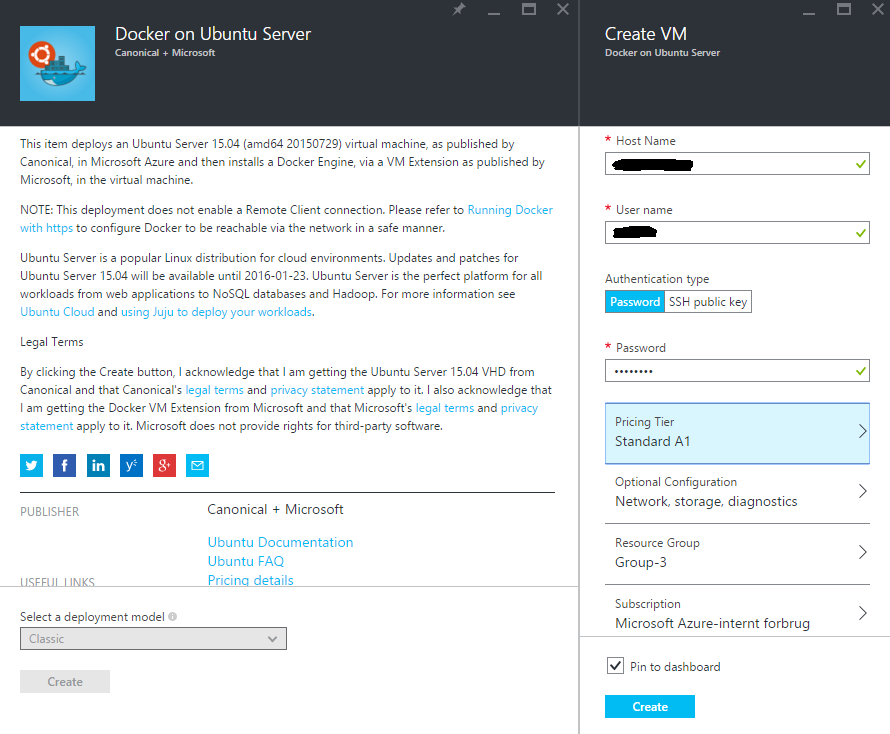
1. Login to the [Azure portal](https://portal.azure.com/)
2. Create a new Marketplace image

[](https://github.com/microsoft-dx/DockerLab/blob/master/Setup/images/SelectCreateMarketplaceVm.png)

1. Search for 'Docker' and select 'Docker on Ubuntu Server'

[](https://github.com/microsoft-dx/DockerLab/blob/master/Setup/images/SearchForDocker.png)

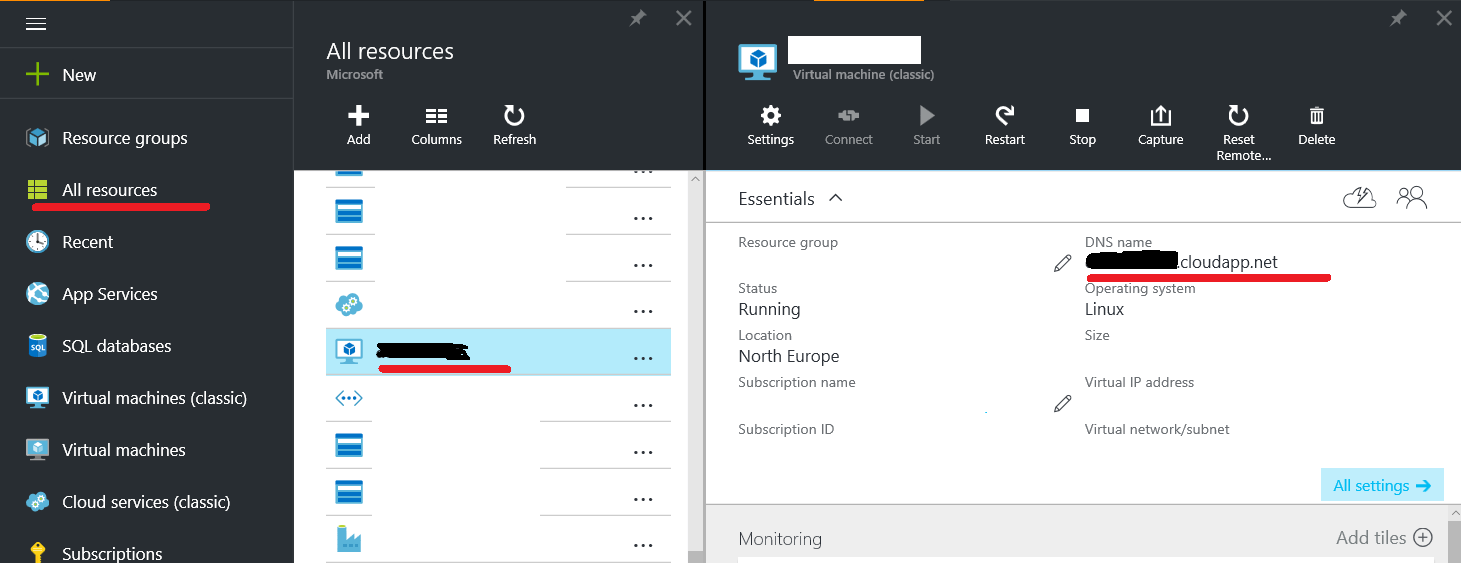
1. Fill in the VM information.
   * The Host Name must be unique, and will determine the DNS name of the Virtual Machine.
   * The User name and Password will be used to access the machine via SSH.
   * For Pricing Tier, select **Standard A1**.

[](https://github.com/microsoft-dx/DockerLab/blob/master/Setup/images/EnterVmSettings.png)

1. Hit **Create** and wait for the Virtual Machine to be provisioned. This usually takes 1-2 minutes to complete.

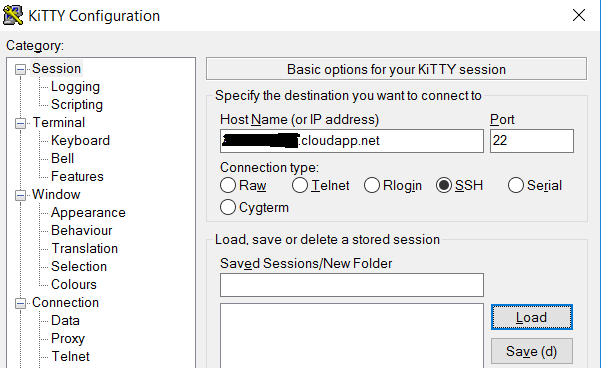
NOTE: The Docker VM image automatically opens for port 22, so it can be accessed via SSH.

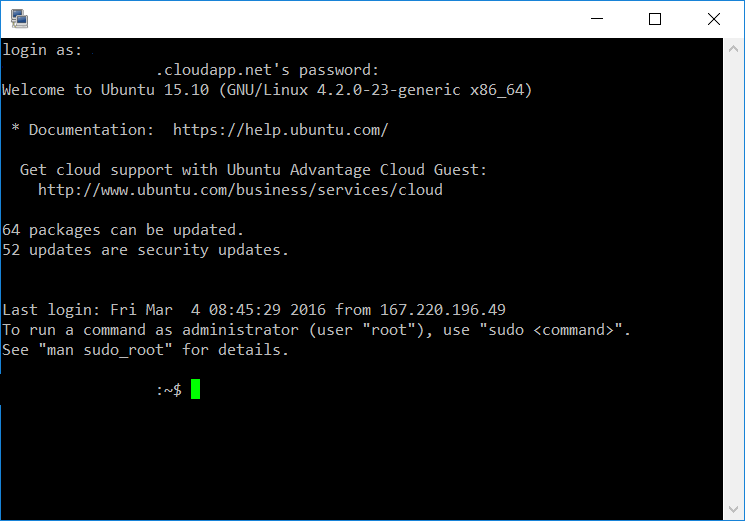
1. To find the automatically assigned DNS name of the VM you just created, navigate to the Settings page of the machine in the Azure Portal, and check for **DNS name**. This is the DNS name you can use for the machine address.

[](https://github.com/microsoft-dx/DockerLab/blob/master/Setup/images/FindDNSName.png)

NOTE: You could also use the Virtual IP address (also called VIP) which is found in the same tab. However, keep in mind that the machine will get a new VIP if it is deallocated by the user at any time. However, the DNS name will be reconfigured to point to the new VIP, therefore you can still use it.

1. Now it's time to connect to the VM. Open your favourite SSH client - on Windows it can be **[PuTTY](http://www.putty.org/)** or **[KiTTY](http://www.9bis.net/kitty/)**.

[](https://github.com/microsoft-dx/DockerLab/blob/master/Setup/images/kitty.png)

[](https://github.com/microsoft-dx/DockerLab/blob/master/Setup/images/kitty2.png)

# Exercise 1 - Basic Docker Commands

Docker allows you to run applications, worlds you create, inside containers. Running an application inside a container takes a single command: docker run.

## Run a Hello world

Let's try it now.

$ docker run ubuntu /bin/echo 'Hello world'

Unable to find image 'ubuntu:latest' locally

latest: Pulling from library/ubuntu

203137e8afd5: Pull complete

2ff1bbbe9310: Pull complete

933ae2486129: Pull complete

a3ed95caeb02: Pull complete

Digest: sha256:1bea66e185d3464fec1abda32ffaf2a11de69833cfcf81bd2b9a5be147776814

Status: Downloaded newer image for ubuntu:latest

Hello world

And you just launched your first container!

So, what just happened? Let's step through what the docker run command did.

First, we specified the docker binary and the command we wanted to execute, run. The docker run combination runs containers.

Next, we specified an image: ubuntu. This is the source of the container we ran. Docker calls this an image. In this case we used the Ubuntu operating system image.

When you specify an image, Docker looks first for the image on your Docker host. If it can't find it then it downloads the image from the public image registry: [Docker Hub](https://hub.docker.com/).

Next we told Docker what command to run inside our new container:

/bin/echo 'Hello world'

When our container was launched, Docker created a new Ubuntu environment and then executed the /bin/echo command inside it. We saw the result on the command line:

Hello world

So what happened to our container after that? Well Docker containers only run as long as the command you specify is active. Here, as soon as Hello world was echoed, the container stopped.

## An interactive container

Let's try the docker run command again, this time specifying a new command to run in our container.

$ docker run -t -i ubuntu /bin/bash

root@af8bae53bdd3:/#

Here we've again specified the docker run command and launched an ubuntu image. But we've also passed in two flags: -tand -i. The -t flag assigns a pseudo-tty or terminal inside our new container and the -i flag allows us to make an interactive connection by grabbing the standard in (STDIN) of the container.

We've also specified a new command for our container to run: /bin/bash. This will launch a Bash shell inside our container.

So now when our container is launched we can see that we've got a command prompt inside it:

root@af8bae53bdd3:/#

Let's try running some commands inside our container:

root@af8bae53bdd3:/# pwd

/

root@af8bae53bdd3:/# ls

bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

You can see we've run the pwd to show our current directory and can see we're in the /root directory. We've also done a directory listing of the root directory which shows us what looks like a typical Linux file system.

You can play around inside this container and when you're done you can use the exit command or enter Ctrl-D to finish.

root@af8bae53bdd3:/# exit

As with our previous container, once the Bash shell process has finished, the container is stopped.

## A daemonized Hello world

Now a container that runs a command and then exits has some uses but it's not overly helpful. Let's create a container that runs as a daemon, like most of the applications we're probably going to run with Docker.

Again we can do this with the docker run command:

$ docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"

1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147

Wait, what? Where's our hello world output? Let's look at what we've run here. It should look pretty familiar. We ran docker run but this time we specified a flag: -d. The -d flag tells Docker to run the container and put it in the background, to daemonize it.

We also specified the same image: ubuntu.

Finally, we specified a command to run:

/bin/sh -c "while true; do echo hello world; sleep 1; done"

This is the (hello) world's silliest daemon: a shell script that echoes hello world forever.

So why aren't we seeing any hello world's? Instead Docker has returned a really long string:

1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147

This really long string is called a container ID. It uniquely identifies a container so we can work with it.

Note: The container ID is a bit long and unwieldy. A bit later, we'll see a shorter ID and ways to name our containers to make working with them easier.

We can use this container ID to see what's happening with our hello world daemon.

Firstly let's make sure our container is running. We can do that with the docker ps command. The docker ps command queries the Docker daemon for information about all the containers it knows about.

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

1e5535038e28 ubuntu /bin/sh -c 'while tr 2 minutes ago Up 1 minute insane\_babbage

Here we can see our daemonized container. The docker ps has returned some useful information about it, starting with a shorter variant of its container ID: 1e5535038e28.

We can also see the image we used to build it, ubuntu, the command it is running, its status and an automatically assigned name, insane\_babbage.

Note: Docker automatically generates names for any containers started. We'll see how to specify your own names a bit later.

Okay, so we now know it's running. But is it doing what we asked it to do? To see this we're going to look inside the container using the docker logs command. Let's use the container name Docker assigned.

$ docker logs insane\_babbage

hello world

hello world

hello world

The docker logs command looks inside the container and returns its standard output: in this case the output of our command hello world.

Awesome! Our daemon is working and we've just created our first Dockerized application!

Now we've established we can create our own containers let's tidy up after ourselves and stop our detached container. To do this we use the docker stop command.

$ docker stop insane\_babbage

insane\_babbage

The docker stop command tells Docker to politely stop the running container. If it succeeds it will return the name of the container it has just stopped.

Let's check it worked with the docker ps command.

$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Excellent. Our container has been stopped.

### References

This exercise borrows and adapts content from the following sources:

1. [The official Docker documentation](https://docs.docker.com/)

# Exercise 2 - Docker Images

Docker images are the basis of containers. Each time you've used docker run you told it which image you wanted. In the previous section of the guide you used Docker images that already exist, for example the ubuntu image.

You also discovered that Docker stores downloaded images on the Docker host. If an image isn't already present on the host then it'll be downloaded from a registry: by default the [Docker Hub](https://hub.docker.com/) Registry.

In this section you're going to explore Docker images a bit more including:

* Managing and working with images locally on your Docker host.
* Creating basic images.
* Uploading images to Docker Hub Registry.

## Listing images on the host

Let's start with listing the images you have locally on our host. You can do this using the docker images command like so:

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

ubuntu 14.04 1d073211c498 3 days ago 187.9 MB

You can see the images you've previously used in the user guide. Each has been downloaded from Docker Hub when you launched a container using that image. When you list images, you get three crucial pieces of information in the listing.

* What repository they came from, for example ubuntu.
* The tags for each image, for example 14.04.
* The image ID of each image.

Tip: You can use a [third-party dockviz tool](https://github.com/justone/dockviz) or the [Image layers site](https://imagelayers.io/) to display visualizations of image data.

A repository potentially holds multiple variants of an image. In the case of our ubuntu image you can see multiple variants covering Ubuntu 10.04, 12.04, 12.10, 13.04, 13.10 and 14.04. Each variant is identified by a tag and you can refer to a tagged image like so:

ubuntu:14.04

So when you run a container you refer to a tagged image like so:

$ docker run -t -i ubuntu:14.04 /bin/bash

If instead you wanted to run an Ubuntu 12.04 image you'd use:

$ docker run -t -i ubuntu:12.04 /bin/bash

If you don't specify a variant, for example you just use ubuntu, then Docker will default to using the ubuntu:latest image.

Tip: You should always specify an image tag, for example ubuntu:14.04. That way, you always know exactly what variant of an image you are using. This is useful for troubleshooting and debugging.

## Getting a new image

So how do you get new images? Well Docker will automatically download any image you use that isn't already present on the Docker host. But this can potentially add some time to the launch of a container. If you want to pre-load an image you can download it using the docker pull command. Suppose you'd like to download the centos image.

$ docker pull centos

Pulling repository centos

b7de3133ff98: Pulling dependent layers

5cc9e91966f7: Pulling fs layer

511136ea3c5a: Download complete

ef52fb1fe610: Download complete

. . .

Status: Downloaded newer image for centos

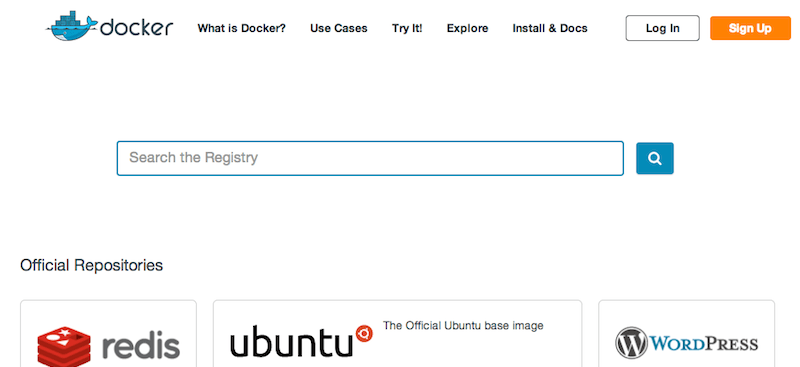
You can see that each layer of the image has been pulled down and now you can run a container from this image and you won't have to wait to download the image.

$ docker run -t -i centos /bin/bash

bash-4.1#

## Finding images

One of the features of Docker is that a lot of people have created Docker images for a variety of purposes. Many of these have been uploaded to [Docker Hub](https://hub.docker.com/). You can search these images on the [Docker Hub](https://hub.docker.com/) website.

[](https://github.com/microsoft-dx/DockerLab/blob/master/Exercise02/images/search.png)

You can also search for images on the command line using the docker search command. Suppose your team wants an image with Ruby and Sinatra installed on which to do our web application development. You can search for a suitable image by using the docker search command to find all the images that contain the term sinatra.

$ docker search sinatra

NAME DESCRIPTION STARS OFFICIAL AUTOMATED

training/sinatra Sinatra training image 0 [OK]

marceldegraaf/sinatra Sinatra test app 0

mattwarren/docker-sinatra-demo 0 [OK]

luisbebop/docker-sinatra-hello-world 0 [OK]

bmorearty/handson-sinatra handson-ruby + Sinatra for Hands on with D... 0

subwiz/sinatra 0

bmorearty/sinatra 0

. . .

You've reviewed the images available to use and you decided to use the training/sinatra image. So far you've seen two types of images repositories, images like ubuntu, which are called base or root images. These base images are provided by Docker Inc and are built, validated and supported. These can be identified by their single word names.

You've also seen user images, for example the training/sinatra image you've chosen. A user image belongs to a member of the Docker community and is built and maintained by them. You can identify user images as they are always prefixed with the user name, here training, of the user that created them.

## Pulling our image

You've identified a suitable image, training/sinatra, and now you can download it using the docker pull command.

$ docker pull training/sinatra

The team can now use this image by running their own containers.

$ docker run -t -i training/sinatra /bin/bash

root@a8cb6ce02d85:/#

## Creating our own images

The team has found the training/sinatra image pretty useful but it's not quite what they need and you need to make some changes to it. There are two ways you can update and create images.

1. You can update a container created from an image and commit the results to an image.
2. You can use a Dockerfile to specify instructions to create an image.

## Updating and committing an image

To update an image you first need to create a container from the image you'd like to update.

$ docker run -t -i training/sinatra /bin/bash

root@0b2616b0e5a8:/#

Note: Take note of the container ID that has been created, 0b2616b0e5a8, as you'll need it in a moment.

Inside our running container let's add the json gem.

Check ruby version

# ruby –version

ruby 1.9.3p484 (2013-11-22 revision 43786) [x86\_64-linux]

We need at least ruby2.x to install our next gem

# sudo apt-get install ruby2.0

Link the ruby2.0 to ruby command, find where is ruby2.0 and then create the symlink

# whereis ruby2.0

ruby2: /usr/bin/ruby2.0

# sudo ln -sf /usr/bin/ruby2.0 /usr/bin/ruby

Install the ruby dev tools

# sudo apt-get install ruby-dev libsqlite3-dev

Install the json gem

# gem install json -v 1.8.1

# exit

Once this has completed let's exit our container using the exit command.

Now you have a container with the change you want to make. You can then commit a copy of this container to an image using the docker commit command.

$ docker commit -m "Added json gem" -a "Kate Smith" \

0b2616b0e5a8 ouruser/sinatra:v2

4f177bd27a9ff0f6dc2a830403925b5360bfe0b93d476f7fc3231110e7f71b1c

Here you've used the docker commit command. You've specified two flags: -m and -a. The -m flag allows us to specify a commit message, much like you would with a commit on a version control system. The -a flag allows us to specify an author for our update.

You've also specified the container you want to create this new image from, 0b2616b0e5a8 (the ID you recorded earlier) and you've specified a target for the image: ouruser/sinatra:v2

Break this target down. It consists of a new user, ouruser, that you're writing this image to. You've also specified the name of the image, here you're keeping the original image name sinatra. Finally you're specifying a tag for the image: v2.

You can then look at our new ouruser/sinatra image using the docker images command.

$ docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

training/sinatra latest 5bc342fa0b91 10 hours ago 446.7 MB

ouruser/sinatra v2 3c59e02ddd1a 10 hours ago 446.7 MB

ouruser/sinatra latest 5db5f8471261 10 hours ago 446.7 MB

To use our new image to create a container you can then:

$ docker run -t -i ouruser/sinatra:v2 /bin/bash

root@78e82f680994:/#

## Building an image from a Dockerfile

Using the docker commit command is a pretty simple way of extending an image but it's a bit cumbersome and it's not easy to share a development process for images amongst a team. Instead you can use a new command, docker build, to build new images from scratch.

To do this you create a Dockerfile that contains a set of instructions that tell Docker how to build our image.

First, create a directory and a Dockerfile.

$ mkdir sinatra

$ cd sinatra

$ touch Dockerfile

Each instruction creates a new layer of the image. Try a simple example now for building your own Sinatra image for your fictitious development team.

Open the Dockerfile with your favorite text editor - for example, nano:

$ nano Dockerfile

Type or paste the following lines inside the Dockerfile:

# This is a comment

FROM ubuntu:14.04

MAINTAINER Kate Smith <ksmith@example.com>

RUN apt-get update && apt-get install -y ruby2.0 ruby-dev

RUN gem install sinatra

Now save the file and close the text editor. If you're using nano, the key combination is CTRL + O to save, and CTRL + X to exit, respectively.

Let's examine what your Dockerfile does. Each instruction prefixes a statement and is capitalized.

INSTRUCTION statement

Note: You use # to indicate a comment

The first instruction FROM tells Docker what the source of our image is, in this case you're basing our new image on an Ubuntu 14.04 image. The instruction uses the MAINTAINER instruction to specify who maintains the new image.

Lastly, you've specified two RUN instructions. A RUN instruction executes a command inside the image, for example installing a package. Here you're updating our APT cache, installing Ruby and RubyGems and then installing the Sinatra gem.

Now let's take our Dockerfile and use the docker build command to build an image.

NOTE: There is a . (dot) at the end of the docker build command, which tells Docker to build from the current folder.

$ docker build -t ouruser/sinatra:v2 .

Sending build context to Docker daemon 2.048 kB

Sending build context to Docker daemon

Step 1 : FROM ubuntu:14.04

---> e54ca5efa2e9

Step 2 : MAINTAINER Kate Smith <ksmith@example.com>

---> Using cache

---> 851baf55332b

Step 3 : RUN apt-get update && apt-get install -y ruby ruby-dev

---> Running in 3a2558904e9b

. . .

---> c55c31703134

Removing intermediate container 3a2558904e9b

Step 4 : RUN gem install sinatra

---> Running in 6b81cb6313e5

. . .

---> 97feabe5d2ed

Removing intermediate container 6b81cb6313e5

Successfully built 97feabe5d2ed

You've specified our docker build command and used the -t flag to identify our new image as belonging to the userouruser, the repository name sinatra and given it the tag v2.

You've also specified the location of our Dockerfile using the . to indicate a Dockerfile in the current directory.

Note: You can also specify a path to a Dockerfile.

Now you can see the build process at work. The first thing Docker does is upload the build context: basically the contents of the directory you're building in. This is done because the Docker daemon does the actual build of the image and it needs the local context to do it.

Next you can see each instruction in the Dockerfile being executed step-by-step. You can see that each step creates a new container, runs the instruction inside that container and then commits that change - just like the docker commit work flow you saw earlier. When all the instructions have executed you're left with the 97feabe5d2ed image (also helpfully tagged asouruser/sinatra:v2) and all intermediate containers will get removed to clean things up.

You can then create a container from our new image.

$ docker run -t -i ouruser/sinatra:v2 /bin/bash

root@8196968dac35:/#

## Setting tags on an image

You can also add a tag to an existing image after you commit or build it. We can do this using the docker tag command. Now, add a new tag to your ouruser/sinatra image.

$ docker tag 5db5f8471261 ouruser/sinatra:devel

The docker tag command takes the ID of the image, here 5db5f8471261, and our user name, the repository name and the new tag.

Now, see your new tag using the docker images command.

$ docker images ouruser/sinatra

REPOSITORY TAG IMAGE ID CREATED SIZE

ouruser/sinatra latest 5db5f8471261 11 hours ago 446.7 MB

ouruser/sinatra devel 5db5f8471261 11 hours ago 446.7 MB

ouruser/sinatra v2 5db5f8471261 11 hours ago 446.7 MB

## Push an image to Docker Hub

Once you've built or created a new image you can push it to Docker Hub using the docker push command. This allows you to share it with others, either publicly, or push it into a private repository.

$ docker push ouruser/sinatra

The push refers to a repository [ouruser/sinatra] (len: 1)

Sending image list

Pushing repository ouruser/sinatra (3 tags)

. . .

## Remove an image from the host

You can also remove images on your Docker host in a way similar to containers using the docker command.

Delete the training/sinatra image as you don't need it anymore.

$ docker rmi training/sinatra

Untagged: training/sinatra:latest

Deleted: 5bc342fa0b91cabf65246837015197eecfa24b2213ed6a51a8974ae250fedd8d

Deleted: ed0fffdcdae5eb2c3a55549857a8be7fc8bc4241fb19ad714364cbfd7a56b22f

Deleted: 5c58979d73ae448df5af1d8142436d81116187a7633082650549c52c3a2418f0

Note: To remove an image from the host, please make sure that there are no containers actively based on it.

### References

This exercise borrows and adapts content from the following sources:

1. [The official Docker documentation](https://docs.docker.com/)