**MODULE 1a**

**Introduction to virt technologies**

**Docker**

***Develop, Ship and Run Any Application, Anywhere***

Docker is a platform for developers and sysadmins to develop, ship, and run applications. Docker lets you quickly assemble applications from components and eliminates the friction that can come when shipping code. Docker lets you get your code tested and deployed into production as fast as possible.

Docker consists of:

* The Docker Engine - our lightweight and powerful open source container virtualization technology combined with a work flow for building and containerizing your applications.
* Docker Hub - SaaS service for sharing and managing your application stacks.

**Why Docker?**

*Faster delivery of your applications*

* If you want your environment to work better. Docker containers, and the work flow that comes with them, help your developers, sysadmins, QA folks, and release engineers work together to get your code into production and make it useful. We've created a standard container format that lets developers care about their applications inside containers while sysadmins and operators can work on running the container in your deployment. This separation of duties streamlines and simplifies the management and deployment of code.
* If you want to make it easy to build new containers, enable rapid iteration of your applications, and increase the visibility of changes. This helps everyone in your organization understand how an application works and how it is built.
* Docker containers are lightweight and fast! Containers have sub-second launch times, reducing the cycle time of development, testing, and deployment.

*Deploy and scale more easily*

* Docker containers run (almost) everywhere. You can deploy containers on desktops, physical servers, virtual machines, into data centers, and up to public and private clouds.
* Since Docker runs on so many platforms, it's easy to move your applications around. You can easily move an application from a testing environment into the cloud and back whenever you need.
* Docker's lightweight containers also make scaling up and down fast and easy. You can quickly launch more containers when needed and then shut them down easily when they're no longer needed.

*Get higher density and run more workloads*

* Docker containers don't need a hypervisor, so you can pack more of them onto your hosts. This means you get more value out of every server and can potentially reduce what you spend on equipment and licenses.

*Faster deployment makes for easier management*

* As Docker speeds up your work flow, it gets easier to make lots of small changes instead of huge, big bang updates. Smaller changes mean reduced risk and more uptime.

**Dockerizing Applications: A "Hello world"**

*So what's this Docker thing all about?*

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

**Hello world**

Let's try it now.

*$ sudo docker run ubuntu:14.04 /bin/echo 'Hello world'*

*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:14.04. This is the source of the container we ran. Docker calls this an image. In this case we used an Ubuntu 14.04 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.

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 14.04 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.

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

*root@af8bae53bdd3:/#*

Here we've again specified the docker run command and launched an ubuntu:14.04 image. But we've also passed in two flags: -t and -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:

*$ sudo docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"*

*1e5535038e285177d5214659a068137486f96ee5c2e85a4ac52dc83f2ebe4147*

Wait what? Where's our "Hello world" 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:14.04.

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 and a bit later on we'll see a shorter ID and some 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.

*$ sudo docker ps*

*CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*

*1e5535038e28 ubuntu:14.04 /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:14.04, the command it is running, its status and an automatically assigned name, insane\_babbage.

**Note:** Docker automatically names any containers you start, a little later on we'll see how you can specify your own names.

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.

*$ sudo 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 daemonized container. To do this we use the docker stop command.

*$ sudo 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.

*$ sudo docker ps*

*CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*

Excellent. Our container has been stopped.

**Working with Containers**

We launched two containers using the docker run command.

* Containers we ran interactively in the foreground.
* One container we ran daemonized in the background.

In the process we learned about several Docker commands:

* docker ps - Lists containers.
* docker logs - Shows us the standard output of a container.
* docker stop - Stops running containers.

## Running a Web Application in Docker

So now we've learnt a bit more about the docker client let's move onto the important stuff: running more containers. So far none of the containers we've run did anything particularly useful though. So

let's build on that experience by running an example web application in Docker.

For our web application we're going to run a Python Flask application. Let's start with a docker run command.

*$ sudo docker run -d -P training/webapp python app.py*

Let's review what our command did. We've specified two flags: -d and -P. We've already seen the

–d flag which tells Docker to run the container in the background. The -P flag is new and tells Docker to map any required network ports inside our container to our host. This lets us view our web application.

We've specified an image: training/webapp. This image is a pre-built image we've created that contains a simple Python Flask web application.

Lastly, we've specified a command for our container to run: python app.py. This launches our web application.

**Note:** You can see more detail on the docker run command in the command reference and the Docker Run Reference.

## Viewing our Web Application Container

Now let's see our running container using the docker ps command.

*$ sudo docker ps -l*

*CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*

*bc533791f3f5 training/webapp:latest python app.py 5 seconds ago Up 2 seconds 0.0.0.0:49155->5000/tcp nostalgic\_morse*

You can see we've specified a new flag, -l, for the docker ps command. This tells the docker ps command to return the details of the last container started.

**Note:** By default, the docker ps command only shows information about running containers. If you want to see stopped containers too use the -a flag.

We can see the same details we saw [when we first Dockerized a container](https://docs.docker.com/userguide/dockerizing) with one important addition in the PORTS column.

*PORTS*

*0.0.0.0:49155->5000/tcp*

When we passed the -P flag to the docker run command Docker mapped any ports exposed in our image to our host.

* **Note:** We'll learn more about how to expose ports in Docker images when we learn how to build images.

In this case Docker has exposed port 5000 (the default Python Flask port) on port 49155.

Network port bindings are very configurable in Docker. In our last example the -P flag is a shortcut for -p 5000 that maps port 5000 inside the container to a high port (from the range 49153 to 65535) on the local Docker host. We can also bind Docker containers to specific ports using the -p flag, for example:

$ sudo docker run -d -p 5000:5000 training/webapp python app.py

This would map port 5000 inside our container to port 5000 on our local host. You might be asking about now: why wouldn't we just want to always use 1:1 port mappings in Docker containers rather than mapping to high ports? Well 1:1 mappings have the constraint of only being able to map one of each port on your local host. Let's say you want to test two Python applications: both bound to port 5000 inside your container. Without Docker's port mapping you could only access one at a time.

So let's now browse to port 49155 in a web browser to see the application.

## Viewing the Web Application's Logs

Let's also find out a bit more about what's happening with our application and use another of the commands we've learnt, docker logs.

*$ sudo docker logs -f nostalgic\_morse*

*\* Running on http://0.0.0.0:5000/*

*10.0.2.2 - - [23/May/2014 20:16:31] "GET / HTTP/1.1" 200 -*

*10.0.2.2 - - [23/May/2014 20:16:31] "GET /favicon.ico HTTP/1.1" 404 -*

This time though we've added a new flag, -f. This causes the docker logs command to act like the tail -f command and watch the container's standard out. We can see here the logs from Flask showing the application running on port 5000 and the access log entries for it.

## Looking at our Web Application Container's processes

In addition to the container's logs we can also examine the processes running inside it using the docker top command.

*$ sudo docker top nostalgic\_morse*

*PID USER COMMAND*

*854 root python app.py*

Here we can see our python app.py command is the only process running inside the container.

## Inspecting our Web Application Container

Lastly, we can take a low-level dive into our Docker container using the docker inspect command. It returns a JSON hash of useful configuration and status information about Docker containers.

*$ sudo docker inspect nostalgic\_morse*

Let's see a sample of that JSON output.

*[{*

*"ID": "bc533791f3f500b280a9626688bc79e342e3ea0d528efe3a86a51ecb28ea20",*

*"Created": "2014-05-26T05:52:40.808952951Z",*

*"Path": "python",*

*"Args": [*

*"app.py"*

*],*

*"Config": {*

*"Hostname": "bc533791f3f5",*

*"Domainname": "",*

*"User": "",*

*. . .*

We can also narrow down the information we want to return by requesting a specific element, for example to return the container's IP address we would:

*$ sudo docker inspect -f '{{ .NetworkSettings.IPAddress }}' nostalgic\_morse*

*172.17.0.5*

## Stopping our Web Application Container

Okay we've seen web application working. Now let's stop it using the docker stop command and the name of our container: nostalgic\_morse.

*$ sudo docker stop nostalgic\_morse*

*nostalgic\_morse*

We can now use the docker ps command to check if the container has been stopped.

*$ sudo docker ps -l*

## Restarting our Web Application Container

Oops! Just after you stopped the container you get a call to say another developer needs the container back. From here you have two choices: you can create a new container or restart the old one. Let's look at starting our previous container back up.

*$ sudo docker start nostalgic\_morse*

*nostalgic\_morse*

Now quickly run docker ps -l again to see the running container is back up or browse to the container's URL to see if the application responds.

**Note:** Also available is the docker restart command that runs a stop and then start on the container.

## Removing our Web Application Container

Your colleague has let you know that they've now finished with the container and won't need it again. So let's remove it using the docker rm command.

*$ sudo docker rm nostalgic\_morse*

*Error: Impossible to remove a running container, please stop it first or use -f*

*2014/05/24 08:12:56 Error: failed to remove one or more containers*

What's happened? We can't actually remove a running container. This protects you from accidentally removing a running container you might need. Let's try this again by stopping the container first.

*$ sudo docker stop nostalgic\_morse*

*nostalgic\_morse*

*$ sudo docker rm nostalgic\_morse*

*nostalgic\_morse*

And now our container is stopped and deleted.

**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 we can use a new command, docker build, to build new images from scratch.

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

Let's create a directory and a Dockerfile first.

$ mkdir sinatra

$ cd sinatra

$ touch Dockerfile

Each instruction creates a new layer of the image. Let's look at a simple example now for building our own Sinatra image for our development team.

# This is a comment

FROM ubuntu:14.04

MAINTAINER Kate Smith <ksmith@example.com>

RUN apt-get update && apt-get install -y ruby ruby-dev

RUN gem install sinatra

Let's look at what our Dockerfile does. Each instruction prefixes a statement and is capitalized.

INSTRUCTION statement

**Note:** We use # to indicate a comment

The first instruction FROM tells Docker what the source of our image is, in this case we're basing our new image on an Ubuntu 14.04 image.

Next we use the MAINTAINER instruction to specify who maintains our new image.

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