

Instructor setup notes

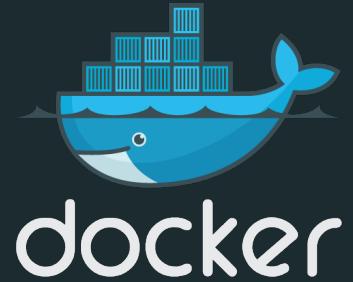
Before the session begins, you need to have done the following. Tasks should ideally be complete the day before training begins

- Setup the student Linux machines. We recommend using AWS or DigitalOcean
- Each student will require 1 machine on your chosen cloud provider
- You may choose to have the Docker pre-installed on the machine but this will mean that students won't be able to go through the installation process themselves
- Make sure that the student account on these machines will have sudo privileges
- You will also require your own Linux instance



Introduction to Docker

The open platform to build, ship and run any applications anywhere

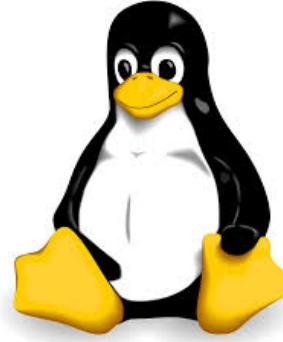


Instructor Intro



Session logistics and pre-requisites

- 2 days including question and exercise time
- Short break every hour
- Lunch break at noon
- Ask questions at anytime



Prerequisites

- No Docker experience required
- You will be provided with a Linux VM
- Be familiar with Linux command line

A screenshot of a terminal window titled "johnnytu@dockertraining: ~". The window shows three blank command-line prompts, each ending with a dollar sign (\$).

```
johnnytu@dockertraining: ~$  
johnnytu@dockertraining: ~$  
johnnytu@dockertraining: ~$
```



Your training environment

- You will be provided with one Ubuntu 14.04 instance
- **Your instructor will provide the login details**
- Majority of exercises will be completed in your master instance

Note: The course materials will assume that the cloud instance is AWS but your instructor may use a different provider such as DigitalOcean etc...



Access your training environment

1. Login to your Amazon AWS instance of Ubuntu using the credentials provided to you by the instructor.
 - a) For MAC or Linux users, use SSH on your terminals
 - b) For PC users, use Putty

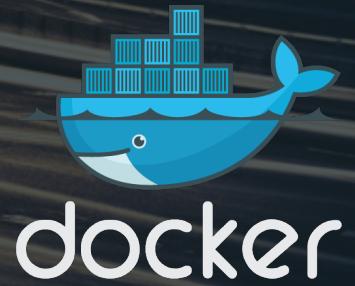


Agenda

- Introduction to containers
- Installing Docker
- Docker concepts and terms
- Introduction to images
- Running and managing containers
- Building images
- Managing and distributing images
- Container volumes
- Container networking
- Docker in continuous integration



Module 1: Introduction to Containers



Module objectives

In this module we will:

- Review the Docker Platform
- Introduce the concept of container based virtualization
- Outline the benefits of containers over virtual machines



What is Docker?

Docker is a platform for developing, shipping and running applications using container technology

- The Docker Platform consists of multiple products/tools
 - Docker Engine
 - Docker Hub
 - Docker Trusted Registry
 - Docker Machine
 - Docker Swarm
 - Docker Compose
 - Kitematic



About Docker, Inc.

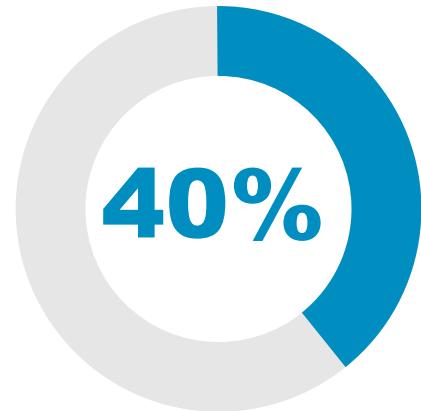
Commercial Docker Solutions

- Integrated solutions to build, ship, run
- Official providers of commercial technical support (Docker, IBM)

Docker Project Sponsor

- Primary contributor and maintainer to Docker open source
- 1B+ Image Downloads
- 1500+ Contributors
- 200,000+ Dockerized Applications

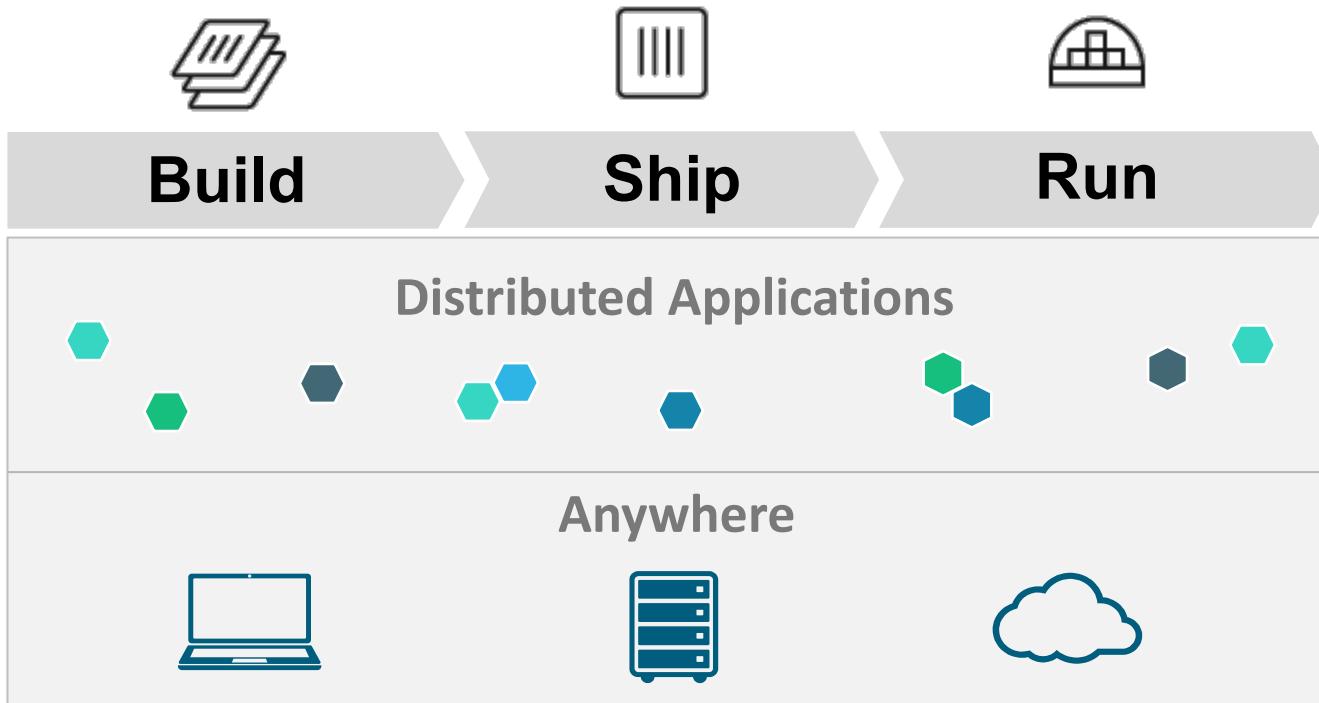
Docker users already running in production



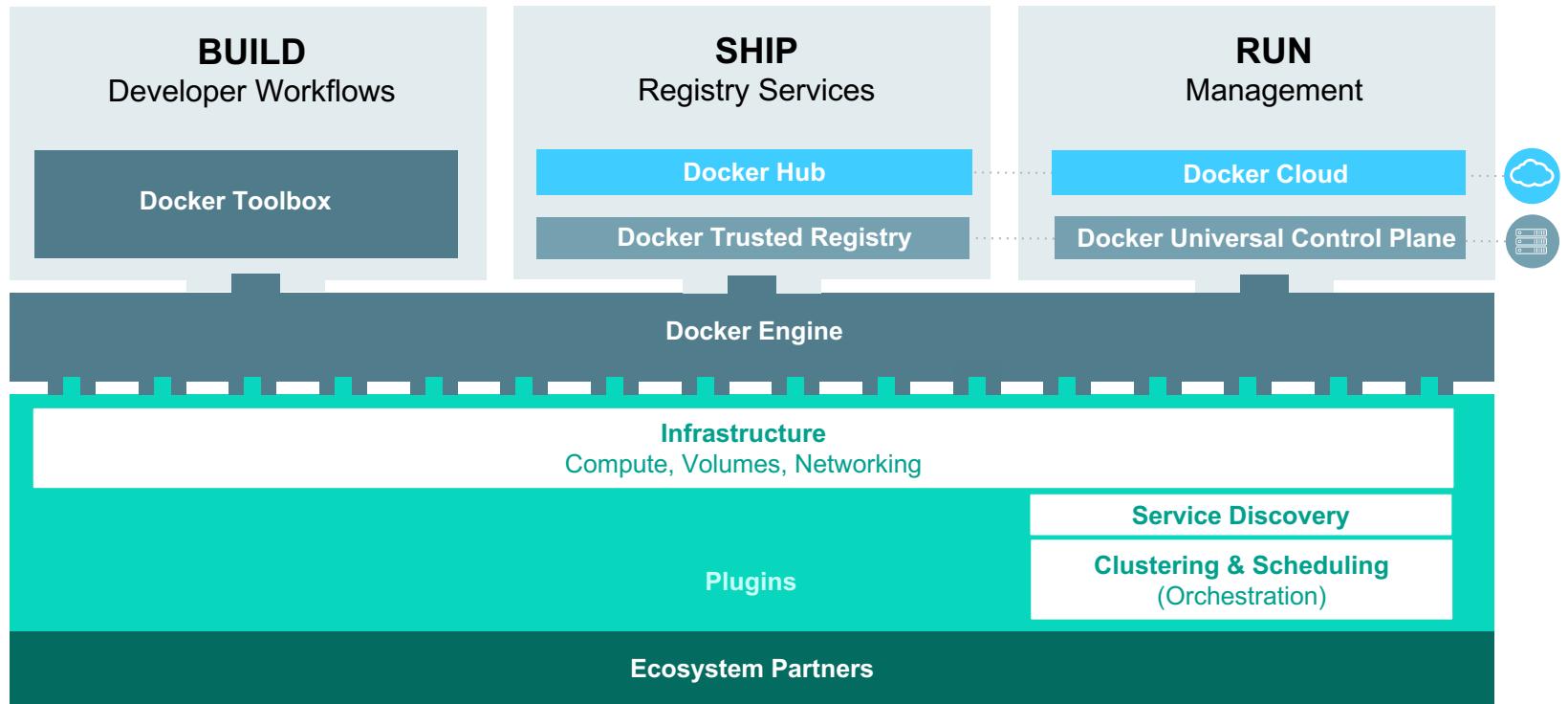
Gerber, Anna. "The State of Containers and the Docker Ecosystem: 2015" O'Reilly, September 2015



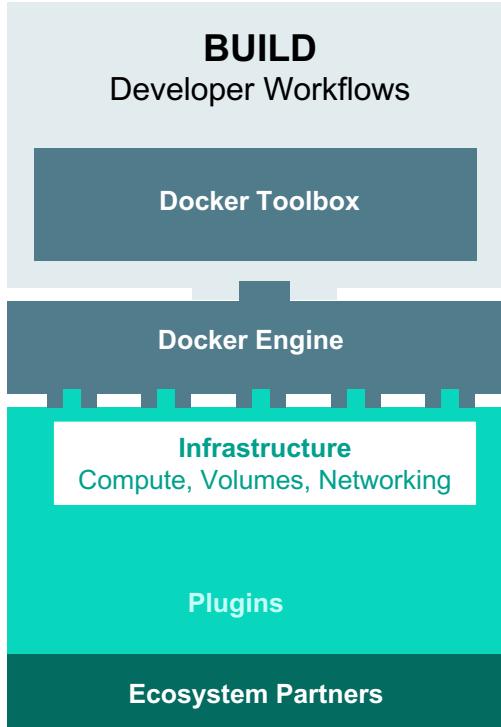
The Docker mission



The Docker Platform



Developer workflows to build Dockerized distributed apps

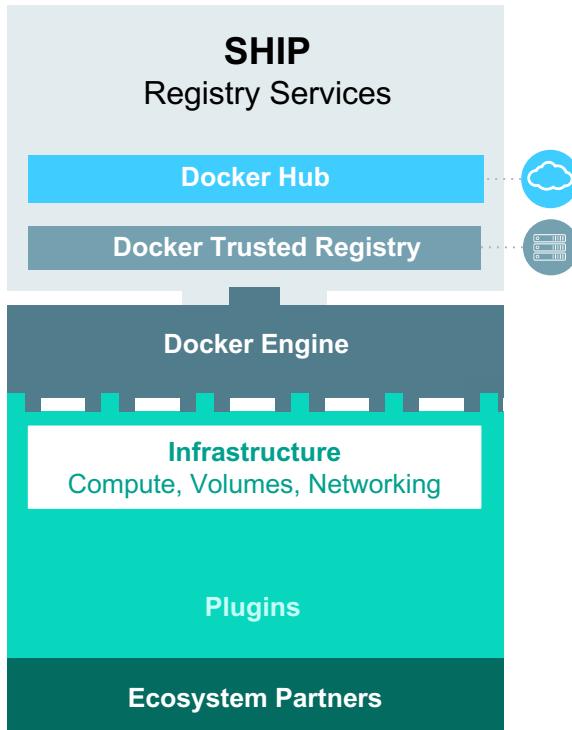


Docker Toolbox installer for Mac and PC sets up the dev environment in a few clicks:

- **Docker Engine** runs locally in a VM
- **Docker Compose** to define multi-container apps
- **Docker Swarm** to cluster Engines and schedule containers
- **Docker Client** for a command line user experience
- **Kitematic** for a graphical user experience (GUI)



Secure content and collaboration between dev and IT ops



- Store images in the **cloud or on-premise**
- **Image Registry** stores all your Docker images in the cloud or in your network
- Control **access and permissions** by user or org
- **Content Trust** to sign images
- **Web UI** to search and browse repos, manage users and settings
- Integrate to CI and CD systems to **automate workflows**

Trusted Registry

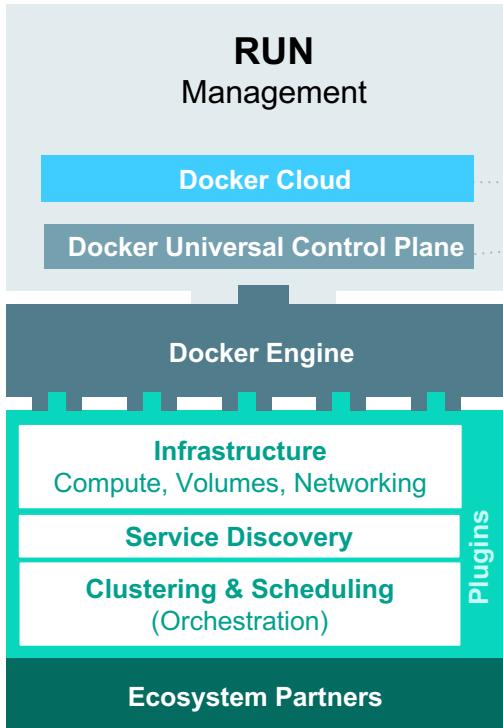
- LDAP/AD integration
- Flexible storage support
- User audit logs
- Soft delete image tags
- Garbage collection

Hub

- Webhooks and Triggers
- Autobuilds



Deploy and manage Dockerized distributed apps anywhere



- Management console in the **cloud** or **on-premise**
- **Manage across hybrid** infrastructure
- **GUI** management dashboards and visibility
- **Deploy and manage containers** and **Compose** apps
- **Deploy and manage clusters** of Engines (Swarm)
- **Integrated management** for images, network, volumes
- **Registry Integration** to Hub or Trusted Registry
- **System metrics** like monitoring, logs, service history

Universal Control Plane

- LDAP/AD integration /RBAC
- High availability
- TLS support

Docker Cloud

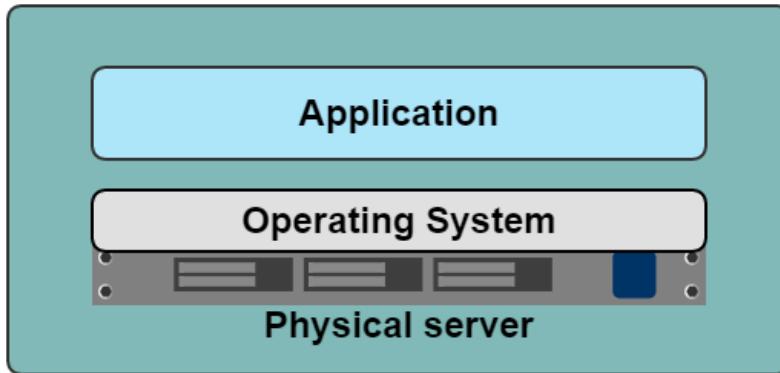
- Secure link to IaaS clouds
- Provision hosts and clusters



A History Lesson

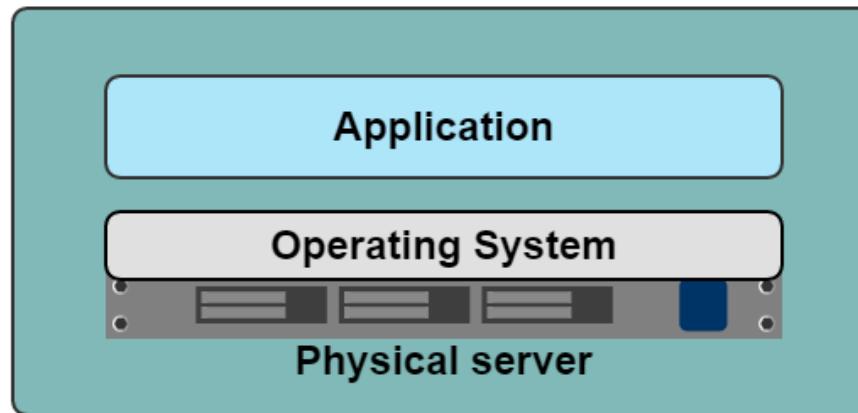
In the Dark Ages

One application on one physical server



Historical limitations of application deployment

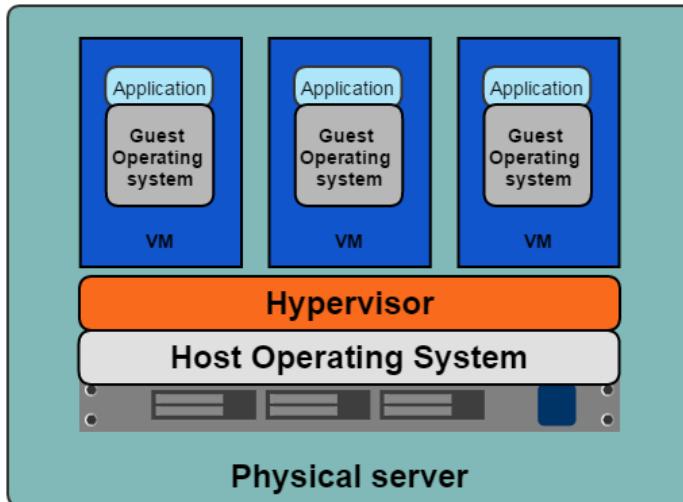
- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Vendor lock in



A History Lesson

Hypervisor-based Virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine (VM)



Benefits of VM's

- Better resource pooling
 - One physical machine divided into multiple virtual machines
- Easier to scale
- VM's in the cloud
 - Rapid elasticity
 - Pay as you go model



Limitations of VM's

- Each VM stills requires
 - CPU allocation
 - Storage
 - RAM
 - An entire guest operating system
- The more VM's you run, the more resources you need
- Guest OS means wasted resources
- Application portability not guaranteed



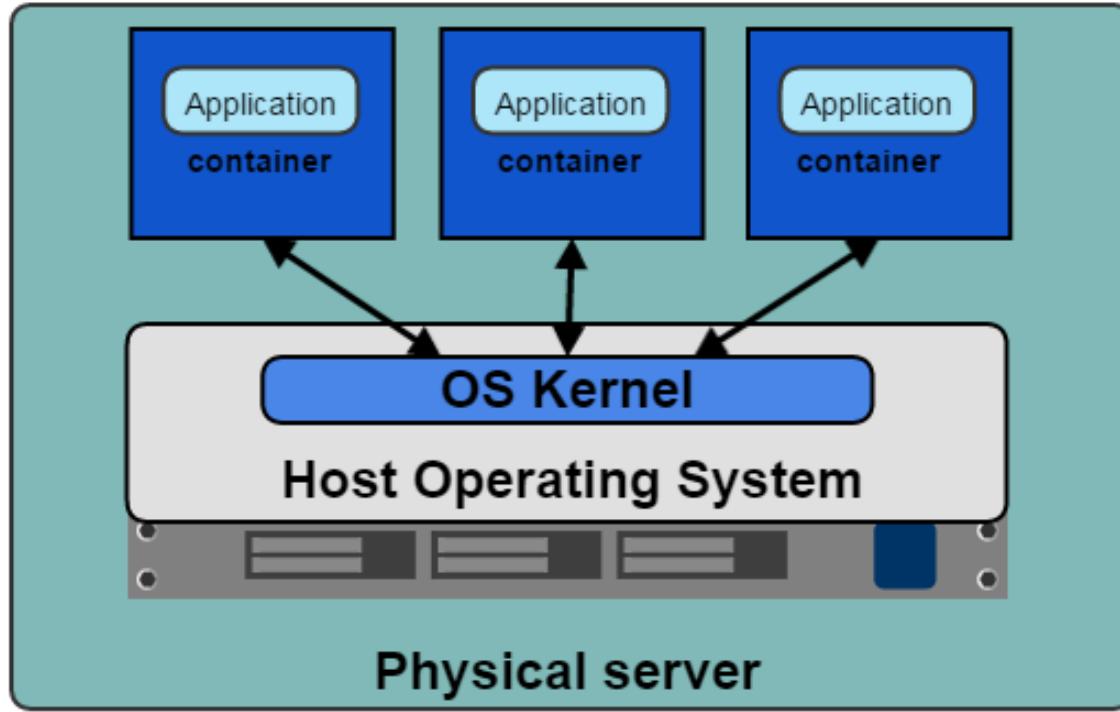
Introducing Containers

Containerization uses the kernel on the host operating system to run multiple root file systems

- Each root file system is called a **container**
- Each container also has its own
 - Processes
 - Memory
 - Devices
 - Network stack



Containers



Containers vs VM's

- Containers are more lightweight
- No need to install guest OS
- Less CPU, RAM, storage space required
- More containers per machine than VMs
- Greater portability

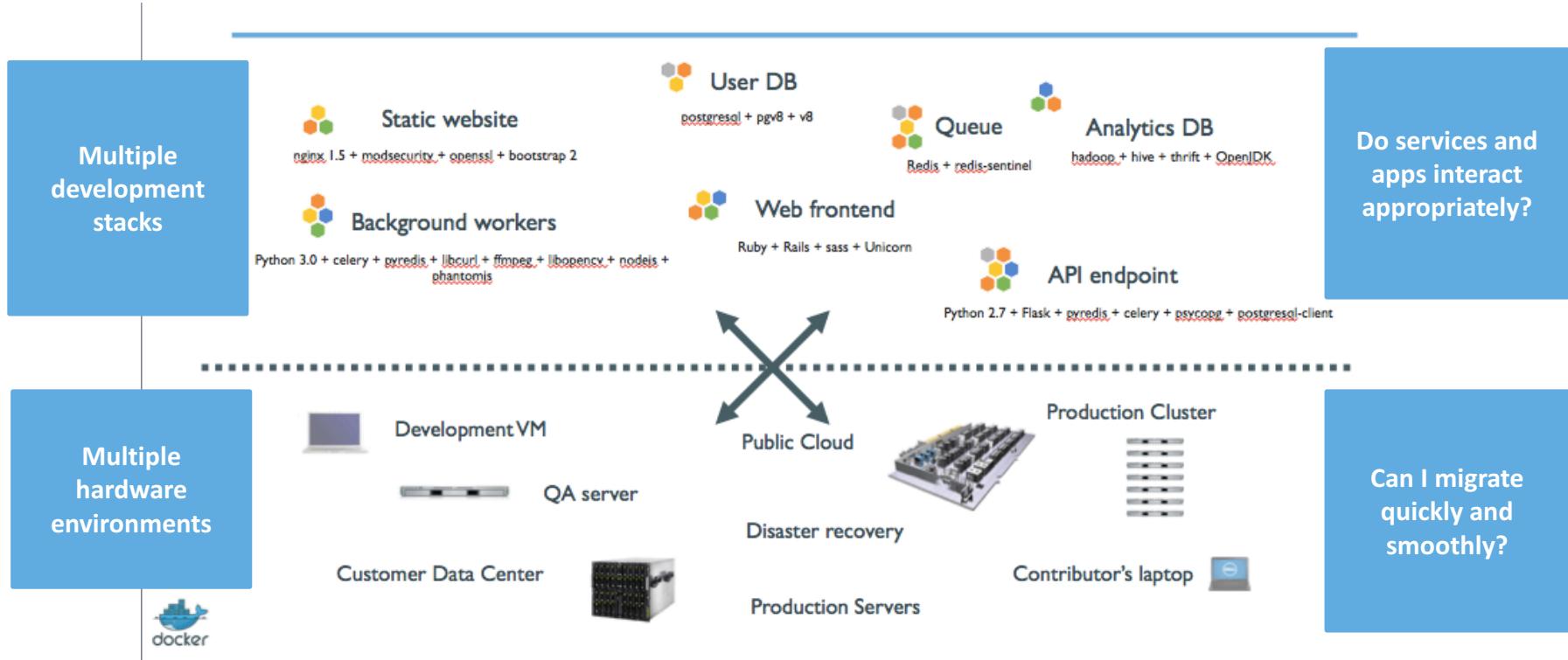


Why use Docker?

- Applications are no longer one big monolithic stack
- Service oriented architecture means there are multiple application stacks that need to be deployed
- Services are decoupled, built iteratively and scaled out
- Deployment can be a complex exercise



The deployment nightmare



The deployment nightmare (cont'd)

	Static website	?	?	?	?	?	?	?						
	Web frontend	?	?	?	?	?	?	?						
	Background workers	?	?	?	?	?	?	?						
	User DB	?	?	?	?	?	?	?						
	Analytics DB	?	?	?	?	?	?	?						
	Queue	?	?	?	?	?	?	?						
	Development VM	Docker icon	QA Server	Docker icon	Single Prod Server	Docker icon	Onsite Cluster	Docker icon	Public Cloud	Docker icon	Contributor's laptop	Docker icon	Customer Servers	Docker icon
														



A shipping analogy

Multiple types
of goods



Do I worry
about how
goods
interact? (i.e.
place coffee
beans next to
spices)

Multiple
methods of
transportation



Can I transport
quickly and
smoothly? (i.e.
unload from
ship onto
train)



The shipping container

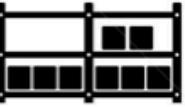
Multiple types of goods



A standard container that is loaded with virtually any goods, and stays sealed until it reaches final delivery.

Do I worry about how goods interact? (i.e. place coffee beans next to spices)

Multiple methods of transportation

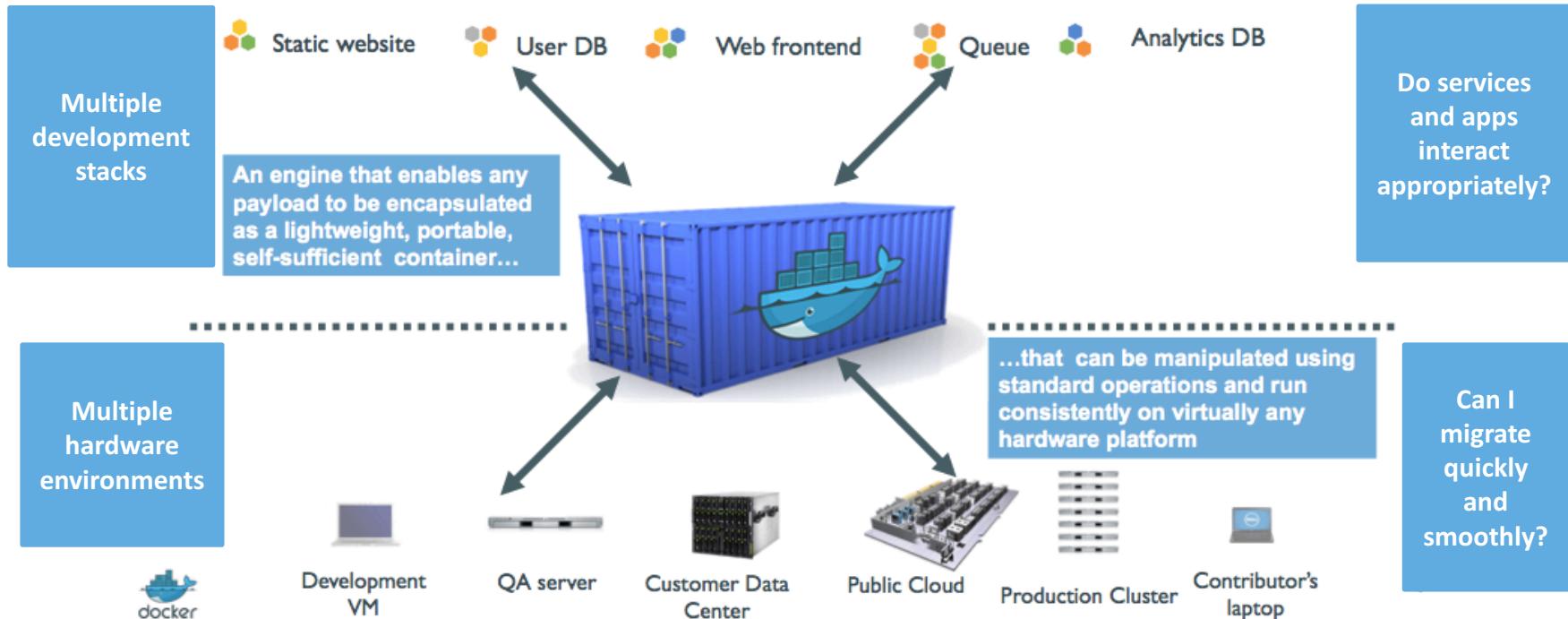


...in between, can be loaded and unloaded, stacked, transported efficiently over long distances, and transferred from one mode of transport to another

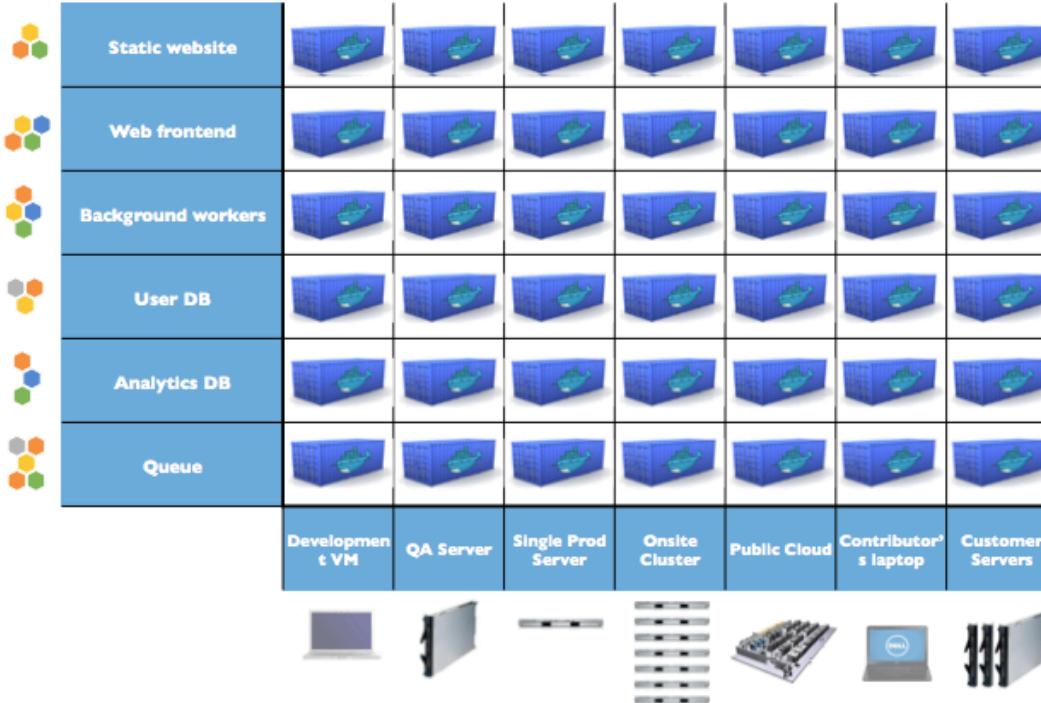
Can I transport quickly and smoothly? (i.e. unload from ship onto train)



Docker containers



Solving the deployment matrix

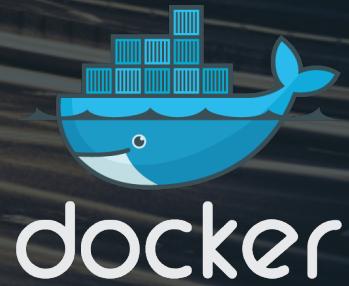


Benefits of Docker

- Separation of concerns
 - Developers focus on building their apps
 - System admins focus on deployment
- Fast development cycle
- Application portability
 - Build in one environment, ship to another
- Scalability
 - Easily spin up new containers if needed
- Run more apps on one host machine



Module 2: Docker Concepts and Terms



Module objectives

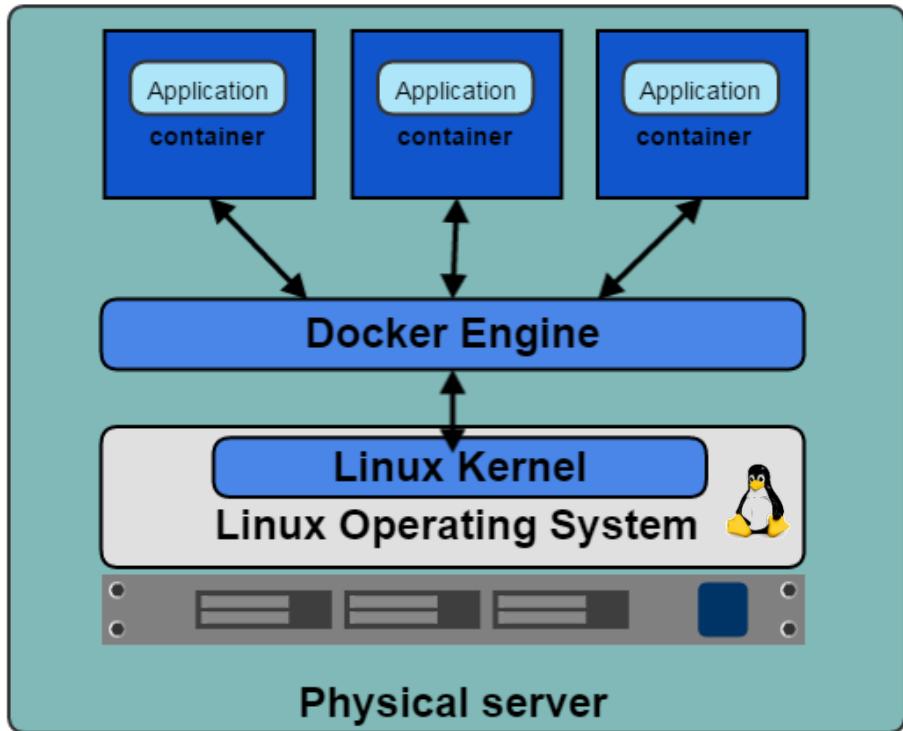
In this module we will:

- Explain all the main Docker concepts including
 - Docker Engine
 - Docker client
 - Containers
 - Images
 - Registry and Repositories
 - Docker Hub
 - Orchestration
 - Docker Toolbox / Machine



Docker and the Linux Kernel

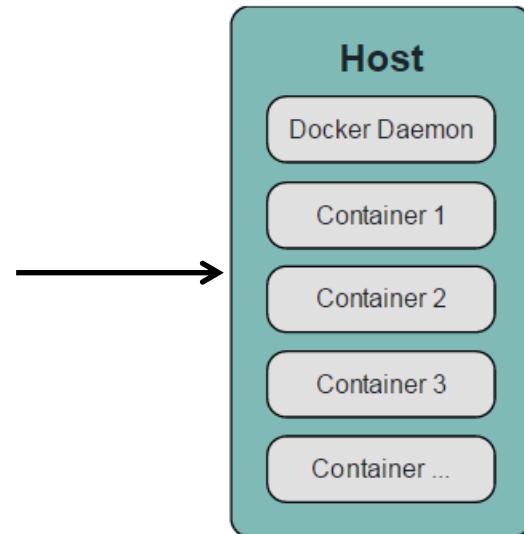
- **Docker Engine** is the program that enables containers to be distributed and run
- Docker Engine uses Linux Kernel namespaces and control groups
- Namespaces give us the isolated workspace



Docker Client and Daemon

- Client / Server architecture
- Client takes user inputs and sends them to the daemon
- Daemon runs and distributes containers
- Client and daemon can run on the same host or on different hosts
- CLI client and GUI (Kitematic)

Client



Checking Client and Daemon Version

- Run
docker version

```
student@DockerTraining:~$ sudo docker version
```

```
Client:
```

```
Version:      1.10.1
API version:  1.22
Go version:   go1.5.3
Git commit:   9e83765
Built:        Thu Feb 11 19:14:21 2016
OS/Arch:      linux/amd64
```

Client

```
Server:
```

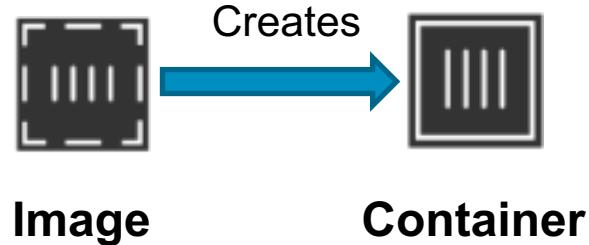
```
Version:      1.10.1
API version:  1.22
Go version:   go1.5.3
Git commit:   9e83765
Built:        Thu Feb 11 20:39:58 2016
OS/Arch:      linux/amd64
```

Daemon

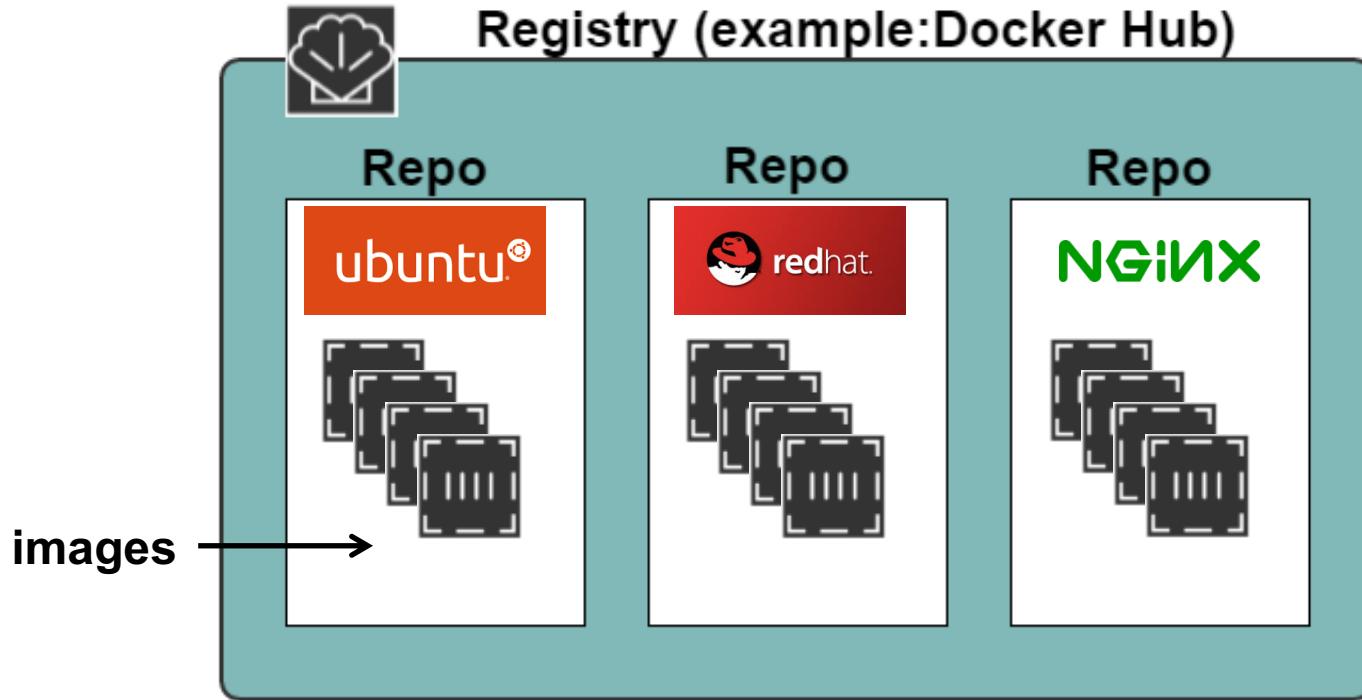


Docker Containers and Images

- Images
 - Read only template used to create containers
 - Built by you or other Docker users
 - Stored in Docker Hub, Docker Trusted Registry or your own Registry
- Containers
 - Isolated application platform
 - Contains everything needed to run your application
 - Based on one or more images



Registry and Repository



Docker Hub

Docker Hub is the public registry that contains a large number of images available for your use

The screenshot shows the Docker Hub interface for exploring official repositories. The top navigation bar includes links for Dashboard, Explore, and Organizations, along with a search bar and user account information. The main section is titled "Explore Official Repositories" and displays four repository cards:

Repository	Description	Stars	Pulls	Action
centos	official	1.4 K	2.2 M	DETAILS
busybox	official	298	36.7 M	DETAILS
ubuntu	official	2.3 K	23.0 M	DETAILS
scratch	official	104	216.3 K	DETAILS



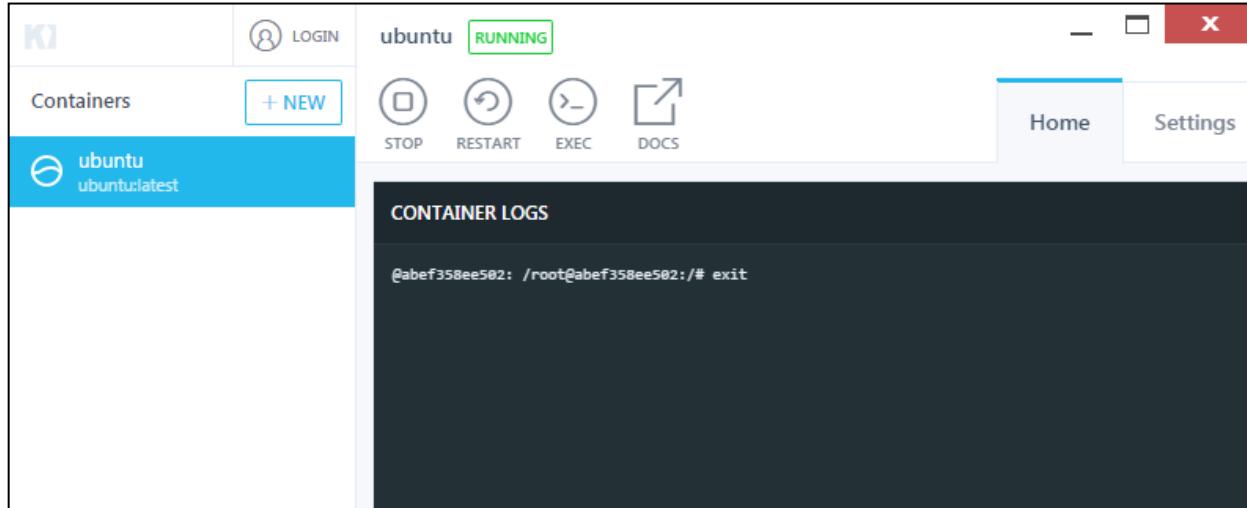
Docker Orchestration

- Three tools for orchestrating distributed applications with Docker
- Docker Machine
 - Tool that provisions Docker hosts and installs the Docker Engine on them
- Docker Swarm
 - Tool that clusters many Engines and schedules containers
- Docker Compose
 - Tool to create and manage multi-container applications



Kitematic

- GUI for Docker
- Start and stop containers
- Connects to Docker Hub to access your images
- Runs on Mac OSX and Windows

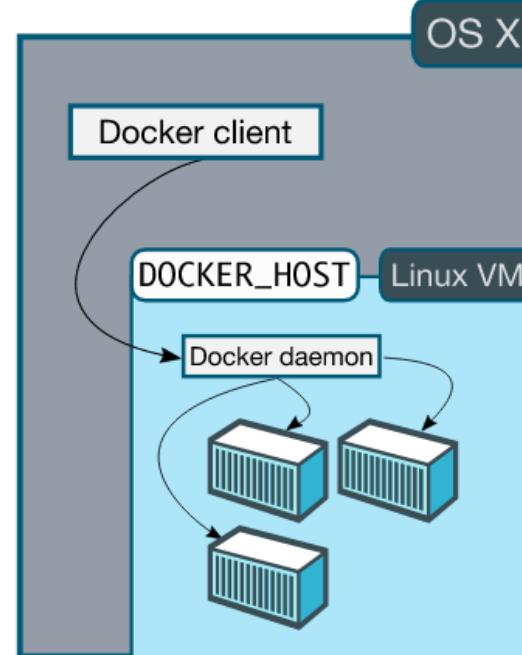
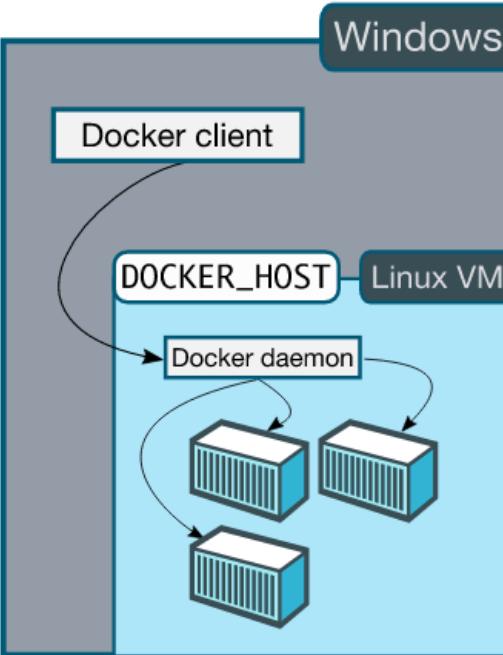
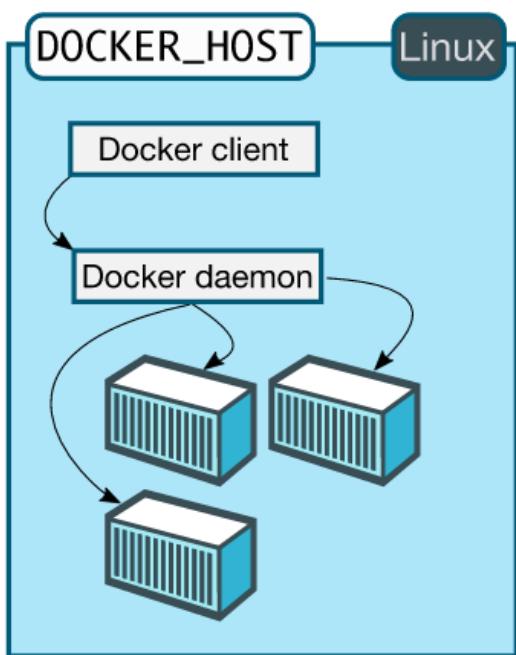


Docker Toolbox

- Docker Engine does not run natively on Windows and Mac OSX
- Need to setup a Linux VM in order to install the Docker Engine
- Docker Toolbox will setup the following
 - Oracle VirtualBox for running a lightweight Linux VM
 - Docker Machine
 - Docker Engine
 - Docker Compose
 - A pre-configured shell for the VM command line
 - Kitematic



Linux vs Windows and Mac OS



Docker Cloud

The best way to deploy and manage Dockerized distributed applications in production



For Devs: Enables DevOps initiatives and self service for deploying and managing distributed apps

For Ops: Easily deploy and manage distributed apps across clouds and private infrastructure

One Platform: Purpose built on Docker for Docker to bring together dev and ops workflows



Docker Universal Control Plane

The first enterprise solution for IT operations control with developer agility and portability of Dockerized applications

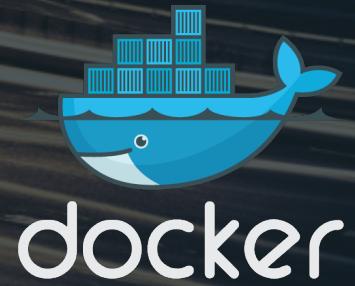
Enterprise ready solution with LDAP/AD integration, on-premise deployment and high availability

For **developer self service** build and deploy apps and **IT Ops** to **deploy and manage** app infrastructure.

Docker native solution with open APIs, pluggable architecture and broadest ecosystem support



Module 3: Installing Docker



Module objectives

In this module we will:

- Outline the ways to install Docker on various Linux operating systems
- Install Docker on our Amazon AWS Ubuntu instance
- Explain how to run Docker without requiring “sudo”
- Install Docker Toolbox on our PC or Mac



Installation options overview

- Docker can be installed in a variety of ways depending on your platform
- Docker managed packaged distributions available for most Linux platforms
 - Ubuntu, CentOS, Fedora, Arch Linux, RHEL, Debian, Gentoo, openSUSE
- Docker managed package is called `docker-engine`
- Old `lxc-docker` and `docker.io` packages are no longer recommended
- Binary download from <https://github.com/docker/docker/releases>



Installing Docker on Ubuntu and Debian

Overview of Steps

1. Add new gpg key
 2. Configure the Docker distribution URL in the file
/etc/apt/sources.list.d/docker.list
 3. Run apt-get update
 4. Run apt-get install docker-engine
-
- **Note:** You must logged in as a privileged user
 - Full instructions at
 - <https://docs.docker.com/engine/installation/ubuntu/>
 - <https://docs.docker.com/engine/installation/debian/>



Adding the gpg key

- Run the command

```
sudo apt-key adv \  
    --keyserver hkp://p80.pool.sks-keyservers.net:80 \  
    --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```



Configuring the docker.list file

- If the file does not already exist, create it inside the /etc/apt/sources.list.d folder
- If the file does exist, delete the existing entries
- Add an entry for your Ubuntu distribution
- The Docker apt distribution URL is <https://apt.dockerproject.org/repo>



Docker distribution URLs for Ubuntu

For Ubuntu Precise 12.04 (LTS)

```
deb https://apt.dockerproject.org/repo ubuntu-precise main
```

For Ubuntu Trusty 14.04 (LTS)

```
deb https://apt.dockerproject.org/repo ubuntu-trusty main
```

For Ubuntu Vivid 15.04

```
deb https://apt.dockerproject.org/repo ubuntu-vivid main
```

For Ubuntu Wily 15.10

```
deb https://apt.dockerproject.org/repo ubuntu-wily main
```



Docker distribution URLs for Debian

Debian Wheezy

```
deb https://apt.dockerproject.org/repo debian-wheezy main
```

Debian Jessie

```
deb https://apt.dockerproject.org/repo debian-jessie main
```

Debian Stretch/Sid

```
deb https://apt.dockerproject.org/repo debian-stretch main
```



Verify the installation

- Run a simple hello-world container
sudo docker run hello-world
- Run sudo docker version to check the version

```
student@DockerTraining:~$ sudo docker version
Client:
  Version:      1.10.1
  API version:  1.22
  Go version:   go1.5.3
  Git commit:   9e83765
  Built:        Thu Feb 11 19:14:21 2016
  OS/Arch:      linux/amd64

Server:
  Version:      1.10.1
  API version:  1.22
  Go version:   go1.5.3
  Git commit:   9e83765
  Built:        Thu Feb 11 20:39:58 2016
  OS/Arch:      linux/amd64
```



EX3.1 – Install Docker on Ubuntu

1. Login to your Ubuntu instance

2. Add the gpg key by running

```
$ sudo apt-key adv \  
--keyserver hkp://p80.pool.sks-keyservers.net:80 \  
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

3. Open the /etc/apt/sources.list.d/docker.list file

4. Delete any existing entries in the file

5. Add the following line

```
deb https://apt.dockerproject.org/repo ubuntu-trusty  
main
```

6. Save and exit the file



EX3.1 – (cont'd)

7. Run `sudo apt-get update`

8. Install Docker by running

```
$ sudo apt-get install docker-engine
```

9. Run the hello-world container to test your installation

```
$ sudo docker run hello-world
```

10. Run `sudo docker version` to check the version of the Docker client and daemon



Installation on CentOS, RHEL, Fedora

- Similar process to Ubuntu but instead of apt-get, we use yum
 1. Update yum packages
\$ sudo yum update
 2. Add the yum repo to /etc/yum.repos.d/docker.repo
 3. Run sudo yum install docker-engine
- Full instructions at
 - <https://docs.docker.com/engine/installation/centos/>
 - <https://docs.docker.com/engine/installation/fedora/>
 - <https://docs.docker.com/engine/installation/rhel/>



The docker group

- To run Docker commands without requiring `sudo`, add your user account to the `docker` group
 - `sudo usermod -aG docker <user>`
 - Also allows for tabbed completion of commands
- Logout and log back in for the changes to take effect
- **Note:** The `docker` group might not have been created on certain distributions. If this is the case, create the group first:
`sudo groupadd docker`
- **Note:** users in the `docker` group essentially have root access. Be aware of the security implications



EX3.2 – Docker group

1. Add your user account to the docker group

```
sudo usermod -aG docker <user>
```

2. Logout of your terminal and log back in for the changes to take effect

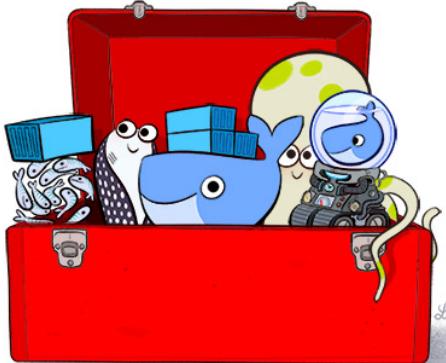
3. Verify that you can run the hello-world container without using sudo
docker run hello-world



Install on Windows and Mac

- Download and install Docker Toolbox from
<https://www.docker.com/toolbox>

Docker Toolbox
[Getting Started Guide \(Mac\)](#) | [Getting Started Guide \(Windows\)](#) | [Contribute to Toolbox](#)



[Download \(Mac\)](#)

[Download \(Windows\)](#)

Compatible with Mac OS X 10.8+ and Windows 7+



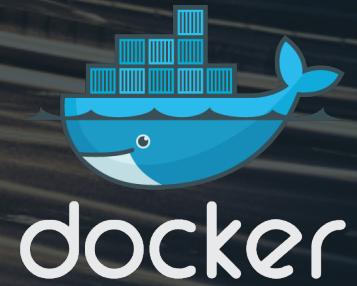
EX3.3 – Install Docker Toolbox

1. Download and Install Docker Toolbox on your PC or MAC
2. Use the Docker Quickstart Terminal to launch a new VM with the Docker Engine installed
3. Verify the installation by running the hello-world container on the Quickstart Terminal
`docker run hello-world`

Note: Students using their own Linux machine will not need to perform this



Module 4: Introduction to Images



Module objectives

In this module we will:

- Learn to search for images on Docker Hub and with the Docker client
- Explain what official repositories are
- Create a Docker Hub account
- Explain the concept of image tags
- Learn to pull images from Docker Hub



Search for Images on Docker Hub

- Lots of Images available for use
- Images reside in various Repositories

The screenshot shows the Docker Hub interface. At the top, there is a navigation bar with a Docker logo, 'Dashboard', 'Explore' (which is highlighted with a red box), 'Organizations', a search bar ('Search' with a magnifying glass icon), 'Create' (with a dropdown arrow), and a user profile ('trainingteam' with a dropdown arrow). Below the navigation bar, the title 'Explore Official Repositories' is displayed. The main content area lists three official repositories in a grid:

Repository	Stars	Pulls	Details
centos official	1.4 K	2.2 M	DETAILS
busybox official	298	36.7 M	DETAILS
ubuntu official	2.3 K	23.0 M	DETAILS



Search for images using Docker client

- Run the docker search command
- Results displayed in table form

```
johnnytu@docker-ubuntu:~$ docker search java
NAME                  DESCRIPTION              STARS      OFFICIAL      AUTOMATED
node                  Node.js is a JavaScript-based platform for... 679        [OK]
java                  Java is a concurrent, class-based, and obj... 180        [OK]
maxexcloo/java        Docker framework container with the Oracle... 6          [OK]
netflixoss/java       Java Base for NetflixOSS container images 4          [OK]
alsanium/java         Java Development Kit (JDK) image for Docker 3          [OK]
andreluiznsilva/java Docker images for java applications 3          [OK]
denvazh/java          Lightweight Java based on Alpine Linux Doc... 2          [OK]
nimmis/java-centos   This is docker images of CentOS 7 with dif... 2          [OK]
isuper/java-oracle    This repository contains all java releases... 2          [OK]
nimmis/java           This is docker images of Ubuntu 14.04 LTS ... 1          [OK]
pallet/java           1          [OK]
isuper/java-openjdk   This repository contains all OpenJDK java ... 1          [OK]
lwieske/java-8        Oracle Java 8 Container 1          [OK]
webratio/java         Java (https://www.java.com/) image 1          [OK]
```



Official repositories

- Official repositories are a certified and curated set of Docker repositories that are promoted on Docker Hub
- Repositories come from vendors such as NGINX, Ubuntu, Red Hat, Redis, etc...
- Images are **supported by their maintainers**, optimised and up to date
- Official repository images are a mixture of
 - Base images for Linux operating systems (Ubuntu, CentOS etc...)
 - Images for popular development tools, programming languages, web and application servers, data stores



Identifying an official repository

- There are a few ways to tell if a repository is official
 - Marked on the OFFICIAL column in the terminal output
 - Repository is labelled “official” on the Docker Hub search results
 - Can filter search results to only display official repositories

Repositories (2452)

All

 java official	368 STARS	840.1 K PULLS	DETAILS
 andreluiznsilva/java public automated build	5 STARS	1.8 K PULLS	DETAILS



EX4.1 - Create a Docker Hub Account

1. Go to <https://hub.docker.com/account/signup/> and signup for an account if you do not already have one.
No credit card details are needed
2. Find your confirmation email and activate your account
3. Browse some of the repositories
4. Search for some images of your favourite dev tools, languages, servers etc...
 - a) (examples: Java, Perl, Maven, Tomcat, NGINX, Apache)



Display Local Images

- Run docker images
- When creating a container Docker will attempt to use a local image first
- If no local image is found, the Docker daemon will look in Docker Hub unless another registry is specified

```
student@DockerTraining:~$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
nginx          latest    ceab60537ad2  9 days ago   132.9 MB
busybox         latest    d7057cb02084  10 days ago  1.096 MB
ubuntu          14.04    91e54dfb1179  6 weeks ago  188.4 MB
hello-world     latest    af340544ed62  8 weeks ago  960 B
student@DockerTraining:~$
```



Image Tags

- Images are specified by **repository:tag**
- The same image may have multiple tags
- The default tag is `latest`
- Look up the repository on Docker Hub to see what tags are available

OFFICIAL REPOSITORY

java ★
Last pushed: 3 days ago

[Repo Info](#) [Tags](#)

Tag	Size
openjdk-8u66-jre	185 MB
openjdk-8u66-jdk	298 MB
openjdk-7u79-jre	141 MB
openjdk-7u79-jdk	241 MB
openjdk-6b36-jre	92 MB
openjdk-6b36-jdk	178 MB
7u79	241 MB



Pulling images

- To download an image from Docker Hub or any registry, use `docker pull` command
- When running a container with the `docker run` command, images are automatically pulled if no local copy is found

Pull the latest image from the Ubuntu repository in Docker Hub

```
docker pull ubuntu
```

Pull the image with tag 12.04 from Ubuntu repository in Docker Hub

```
docker pull ubuntu:12.04
```

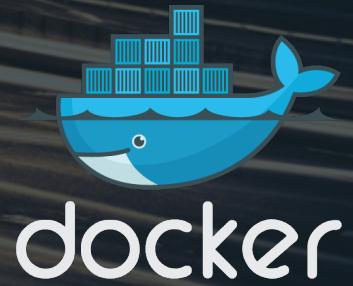


EX4.2 – Display local images

1. Pull a busybox image from Docker Hub
`docker pull busybox`
2. Display your local images and verify that the image is present
`docker images`



Module 5: Running and Managing Containers



Module objectives

In this module we will

- Learn how to launch containers with the `docker run` command
- Explore different methods of accessing a container
- Explain how container processes work
- Learn how to stop and start containers
- Learn how to check container logs
- Learn how to find your containers with the `docker ps` command
- Learn how to display information about a container



Container lifecycle

- Basic lifecycle of a Docker container
 - Create container from image
 - Run container with a specified process
 - Process finishes and container stops
 - Destroy container
- More advanced lifecycle
 - Create container from image
 - Run container with a specified process
 - Interact and perform other actions inside the container
 - Stop the container
 - Restart the container



Creating and running a Container

- Use docker run command
- The docker run command actually does two things
 - Creates the container using the image we specify
 - Runs the container
- Syntax
docker run [options] [image] [command] [args]
- Image is specified with repository:tag

Examples

```
docker run ubuntu:14.04 echo "Hello World"
```

```
docker run ubuntu ps ax
```



EX5.1 - Run a Simple Container

1. On your terminal type

```
docker run ubuntu:14.04 echo "hello world"
```

2. Observe the output

3. Then type

```
docker run ubuntu:14.04 ps -ef
```

4. Observe the output

5. Notice the much faster execution time compared to the first container that was run. This is due to the fact that Docker now has the Ubuntu 14.04 image locally and thus does not need to download the image



Find your Containers

- Use `docker ps` to list running containers
- The `-a` flag to list all containers (includes containers that are stopped)

```
johnnytu@docker-ubuntu:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             ...
27df74c91cad      ubuntu:14.04       "ps -a"            ...
90d52e1c6ccc      ubuntu:14.04       "echo 'hello world'" ...
49c31eb487ab      hello-world:latest  "/hello"           ...
                                                               ...
                                                               NAMES
                                                               lonely_poincare
                                                               elegant_bohr
                                                               agitated_sinoussi
```



EX5.2 – List containers

1. List your running containers. What can you observe?

```
docker ps
```

2. List all your containers, including ones that have stopped

```
docker ps -a
```



Container with Terminal

- Use `-i` and `-t` flags with `docker run`
- The `-i` flag tells docker to connect to STDIN on the container
- The `-t` flag specifies to get a pseudo-terminal
- **Note:** You need to run a terminal process as your command (e.g. `bash`)

Example

```
docker run -i -t ubuntu:latest bash
```



Exit the Terminal

- Type `exit` to quit the terminal and return to your host terminal
- Exiting the terminal will shutdown the container
- To exit the terminal without a shutdown, hit `CTRL + P + Q` together



EX5.3 - Terminal Access

1. Create a container using the ubuntu 14.04 image and connect to STDIN and a terminal

```
docker run -i -t ubuntu:14.04 bash
```

2. In your container, create a file called “test”

```
touch test
```

3. Run ls to verify the presence of the file

4. Exit the container

```
exit
```

5. Notice how the container shut down

6. Once again run:

```
docker run -i -t ubuntu:14.04 bash
```

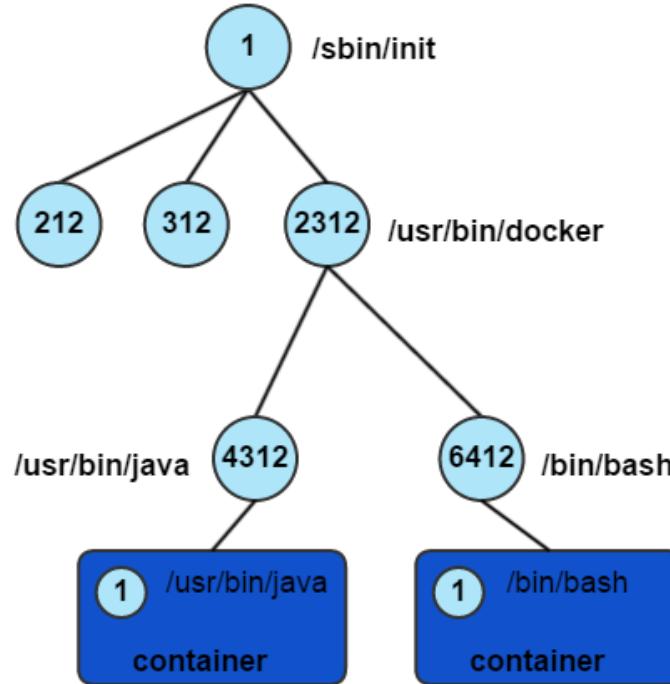
7. Run ls and try to find the “test” file

8. Notice how it does not exist



Container Processes

- A container only runs as long as the process from your specified `docker run` command is running
- Your command's process is always PID 1 inside the container



Container ID

- Containers can be specified using their ID or name
- Long ID and short ID
- Short ID and name can be obtained using `docker ps` command to list containers
- Long ID obtained by inspecting a container or by using the `--no-trunc` flag on the `docker ps` command.
`docker ps -a --no-trunc`



docker ps command

- To view only the container ID's (displays short ID)

```
docker ps -q
```

- To view the last container that was started

```
docker ps -l
```

```
johnnytu@docker-ubuntu:~$ docker ps -l
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              ...   NAMES
9845b1ce9b42        ubuntu:latest      "ping 127.0.0.1 -c 5"   2 days ago         Exited (0) 2 days ago ...   sick_curie
```



docker ps command

- Combining flags to list all containers with only their short ID
docker ps -aq
- Combining flags to list the short ID of the last container started
docker ps -lq

```
johnnytu@docker-ubuntu:~$ docker ps -aq
9845b1ce9b42
0b64e9f387cf
bab6e7c20650
e5c86a4a4bc4
a8d7408fe588
ca6f7f4c3487
feb4656f106d
e330a26c5299
```



docker ps filtering

- Use the `--filter` flag to specify filtering conditions
- Currently you can filter based on the container's exit code and status
- Status can be one of
 - Restarting
 - Running
 - Exited
 - Paused
- To specify multiple conditions, pass multiple `--filter` flags



docker ps filtering examples

- List containers with an exit code of 1 (exited with error)

```
johnnytu@docker-ubuntu:~$ docker ps -a --filter "Exited=1"
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS      NAMES
27df74c91cad      ubuntu:14.04       "ps -a"           4 days ago        Exited (1) 4 days ago          lonely_poincare
```



Running in Detached Mode

- Also known as running in the background or as a daemon
- Use `-d` flag
- To observe output use `docker logs [container id]`

Create a centos container and run the ping command to ping the container itself 50 times

```
docker run -d centos:7 ping 127.0.0.1 -c 50
```



EX5.4 – Run in detached mode

1. Run

```
docker run -d centos:7 ping 127.0.0.1 -c 50
```

2. List your containers by running docker ps

3. Notice the centos container running

4. Wait for minute and run docker ps again

5. You may notice that the container has stopped (if it is still running, wait a bit longer). Why do you think the container has stopped?



A More Practical Container

- Run a web application inside a container
- The `-P` flag to map container ports to host ports

Create a container using the `nginx` image, run in detached mode and map the `nginx` ports to the host port

```
docker run -d -P nginx
```



EX5.5 - Web Application Container

1. Run

```
docker run -d -P nginx
```

2. Check your image details by running

```
docker ps
```

3. Notice the port mapping.

```
johnnytu@docker-ubuntu:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             ...
f7eb3e507e46        nginx:latest       "nginx -g 'daemon off; ..."
                                                               PORTS
                                                               NAMES
                                                               0.0.0.0:32768->80/tcp, 0.0.0.0:32769->443/tcp   happy_bohr
```

4. Go to <your linux server url>:<port number> and verify that you can see the NGINX welcome page



EX5.6 – Check container process

1. Run an Ubuntu container with bash as the process. Remember to use `-it` to gain terminal access
2. Run `ps -ef` and check the PID number of the bash process
3. Exit the container without shutting it down
`CTRL + P + Q`
4. Run `ps -ef` on your host and look for the bash processes
`ps -ef | grep bash`
5. Note down the parent PID of each bash process
6. Find the processes with the parent PID numbers you noted down
7. What do you notice?



Attaching to a container

- Attaching a client to a container will bring a container which is running in the background into the foreground
- The containers PID 1 process output will be displayed on your terminal
- Use `docker attach` command and specify the container ID or name
- **Warning:** Attaching to containers is error prone because if you hit `CTRL + C` by accident, you will stop the process and therefore stop the container

```
johnnytu@docker-ubuntu:~$ docker run -d -it ubuntu ping 127.0.0.1 -c 50
9845b1ce9b429388cd937debba19e630a71b1c942341f10f06ea27d6c500579a
johnnytu@docker-ubuntu:~$ docker attach 9845b1ce9b429388
64 bytes from 127.0.0.1: icmp_seq=12 ttl=64 time=0.086 ms
64 bytes from 127.0.0.1: icmp_seq=13 ttl=64 time=0.092 ms
64 bytes from 127.0.0.1: icmp_seq=14 ttl=64 time=0.050 ms
64 bytes from 127.0.0.1: icmp_seq=15 ttl=64 time=0.065 ms
```



Detaching from a container

- Hit CTRL + P + Q together on your terminal
- Only works if the following two conditions are met
 - The container standard input is connected
 - The container has been started with a terminal
 - For example: `docker run -i -t ubuntu`
- Hitting CTRL + C will terminate the process, thus shutting down the container



EX5.7 – Attach and detach

1. Run

```
docker run -d ubuntu ping 127.0.0.1 -c 50
```

2. Attach to the newly created container and observe the output

3. Try and detach from the container. Notice how you can only exit with CTRL + C

4. Check your running containers

5. Run another container

```
docker run -d -it ubuntu ping 127.0.0.1 -c 50
```

6. Attach to the newly created container

7. Detach from the container with CTRL + P + Q

8. List your running containers



Docker exec command

- docker exec command allows us to execute additional processes inside a container
- Typically used to gain command line access
- docker exec -i -t [container ID] bash
- Exiting from the terminal will not terminate the container



EX5.8 – docker exec

1. Run an NGINX container in the background
`docker run -d nginx`
2. Use `docker exec` to start a bash terminal in the container
`docker exec -it <container id> bash`
3. Exit the container terminal
4. Verify that your container is still running
`docker ps`



Inspecting container logs

- Container PID 1 process output can be viewed with `docker logs` command
- Will show whatever PID 1 writes to stdout and stderr
- Displays the entire log output from the time the container was created

View the output of the containers PID 1 process

```
docker logs <container name>
```



Tailing container logs

- We can specify to only show the last “x” number of lines from the logs
- Use `--tail` option and specify the number of lines
- Use the `--follow` option or `-f` to get a streaming output from the log

Show the last 5 lines from the container log

```
docker logs --tail 5 <container ID>
```

Show the last 5 lines and follow the log

```
docker logs --tail 5 -f <container ID>
```



EX5.9 – container logs

1. Start a new Ubuntu container and run the ping command to ping 127.0.0.1 and have it repeat 100 times

```
docker run -d ubuntu:14.04 ping 127.0.0.1 -c 100
```

2. Inspect the container log, run this command a few times

```
docker logs <container id>
```

3. Then inspect and follow the container log

```
docker logs -f <container id>
```

4. Hit CTRL + C to stop following the log

5. Now follow the log again but trim the output to start from the last 10 lines

```
docker logs --tail 10 -f <container id>
```



Stopping a container

- Two commands we can use
 - docker stop
 - docker kill
- docker stop sends a SIGTERM to the main container process
 - Process then receives a SIGKILL after a grace period
 - Grace period can be specified with -t flag (default is 10 seconds)
- docker kill sends a SIGKILL immediately to the main container process



Restarting a container

- Use `docker start` to restart a container that has been stopped
- Container will start using the same options and command specified previously
- Can attach to the container with `-a` flag

Start a stopped container and attach to the process that it is running

```
docker start -a <container ID>
```



EX5.10 – Stop and Start container

1. Run a tomcat container in detached mode

```
docker run -d tomcat
```

2. Inspect and follow the container log

```
docker logs -f <container id>
```

3. Stop the container

```
docker stop <container id>
```

4. Start the container again and re-attach to it

```
docker start -a <container id>
```



EX5.10 – (cont'd)

5. Detach and stop the container

CTRL + C

6. Restart the container again and follow the log output

docker start <container id>

docker logs -f <container id>

7. Notice how there are so many log lines it is difficult to follow

8. Hit CTRL + C to stop following the log

9. Now inspect the container log again but this time, trim the output to only show the last 10 lines and follow it

docker logs --tail 10 -f <container id>



Inspecting a container

- docker inspect command displays all the details about a container
- Outputs details in JSON array

```
student@DockerTraining:~$ docker inspect serene_kilby
[
{
  "Id": "7061518968d429cb9eb0b47a4ef56f64a8ba24b72ae5e6956c12c66e4c5428d4",
  "Created": "2015-10-02T04:34:24.420481768Z",
  "Path": "nginx",
  "Args": [
    "-g",
    "daemon off;"
  ],
  "State": {
    "Running": true,
    "Paused": false,
    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 20998,
    "ExitCode": 0,
    "Error": "",
    "StartedAt": "2015-10-02T04:34:24.572088631Z",
    "FinishedAt": "0001-01-01T00:00:00Z"
  },
  "Config": {
    "Image": "nginx:latest",
    "Cmd": [
      "nginx"
    ],
    "ExposedPorts": {
      "80/tcp": {}
    },
    "Labels": {}
  }
}
```



EX5.11 – Inspect container properties

1. Inspect your tomcat container from exercise 5.10
`docker inspect <container id>`
2. Look through the JSON output and look for the container IP address and long ID



Finding a specific property

- You can pipe the output of docker inspect to grep and use it to search a particular container property.
- Example

```
docker inspect <container name> | grep IPAddress
```

```
johnnytu@docker-ubuntu:~$ docker inspect jolly_davinci | grep IPAddress
    "IPAddress": "172.17.0.24",
```



Formatting docker inspect output

- Piping the output to `grep` is simple, but not the most effective method of formatting
- Use the `--format` option of the `docker inspect` command
- Format option uses Go's text/template package
<http://golang.org/pkg/text/template/>



docker inspect formatting syntax

- docker inspect --format='{{.<field1>.<field2>}}' \
 <container id>
- Field names are case sensitive
- Example – to get the value of the Pid field
 docker inspect --format='{{.State.Pid}}' <container id>

```
student@DockerTraining:~$ docker inspect serene_kilby
[
{
  "Id": "7061518968d429cb9eb0b47a4ef56f64a8ba24b72ae5e6956c12c66e4c5428d",
  "Created": "2015-10-02T04:34:24.420481768Z",
  "Path": "nginx",
  "Args": [
    "-g",
    "daemon off;"
  ],
  "State": {
    "Running": true,
    "Paused": false,
    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 20998,
  }
}
```



EX5.12 – Format inspect output

1. Inspect your tomcat container and pipe the output to grep. Grep for “IPAddress”

```
docker inspect <container name> | grep IPAddress
```

2. Now try and grep for “Cmd”. Notice how the output does not give us what we want

```
docker inspect <container name> | grep Cmd
```

3. Use the --format option and grab the output of the Cmd field

```
docker inspect --format='{{.Config.Cmd}}' \
    <container name>
```

4. Use the --format option and grab the output of the IPAddress field

```
docker inspect --format='{{.NetworkSettings.IPAddress}}' \
    <container name>
```

5. Repeat step 4, trying different fields of your choice



Inspecting a whole JSON object

- When you want to output all the fields of a JSON object you need to use the Go template's JSON function

```
student@DockerTraining:~$ docker inspect serene_kilby
[
{
  "Id": "7061518968d429cb9eb0b47a4ef56f64a8ba24b72ae5e6956c12c66e4c5428d4",
  "Created": "2015-10-02T04:34:24.420481768Z",
  "Path": "nginx",
  "Args": [
    "-g",
    "daemon off;"
  ],
  "State": {
    "Running": true,
    "Paused": false,
    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 20998,
    "ExitCode": 0
  }
}
```

Incorrect approach

```
docker inspect --format='{{.State}}' <container name>
```

Correct approach

```
docker inspect --format='{{json .State}}' <container name>
```



EX5.13 – More inspection

1. Inspect your tomcat container and take note of the output of the Config object
2. Format the output of your docker inspect command to only display the Config object without using the JSON function
`docker inspect --format='{{ .Config }}' <container name>`
3. Notice how the output is not correct
4. Now apply the json function in your formatting and compare the difference
`docker inspect --format='{{json .Config}}' <container name>`
5. Pick another JSON object from the full docker inspect output and repeat the same steps as above



Deleting containers

- Can only delete containers that have been stopped
- Use `docker rm` command
- Specify the container ID or name
- To delete a container that is still running, use `-f` option

```
docker rm -f <container ID>
```



Delete all containers

- Use `docker ps -aq` to list the id's of all containers
- Feed the output into `docker rm` command
- Output will print an error message for containers that are still running

Delete all containers that are stopped

```
docker rm $(docker ps -aq)
```

Delete all containers (including the ones still running)

```
docker rm -f $(docker ps -aq)
```



EX5.14 – Delete containers

1. List all your stopped containers

```
docker ps --filter='status=exited'
```

2. Delete one container using it's short ID

```
docker rm <container id>
```

3. Delete the latest container that was run (Note that this will fail if the container is still running)

```
docker rm $(docker ps -ql)
```

4. Delete all stopped containers

```
docker rm $(docker ps -aq)
```

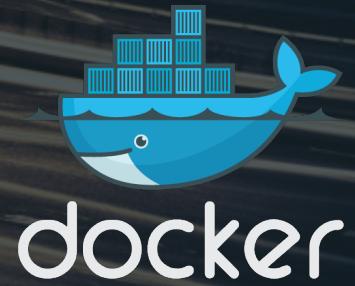


Module Summary

- Containers can be run in the foreground and background
- A container only runs as long as the process we specified during creation is running
- The container log is the output of its PID 1 process
- Key commands we learnt
 - docker run
 - docker ps
 - docker logs
 - docker inspect



Module 6: Building Images



Module objectives

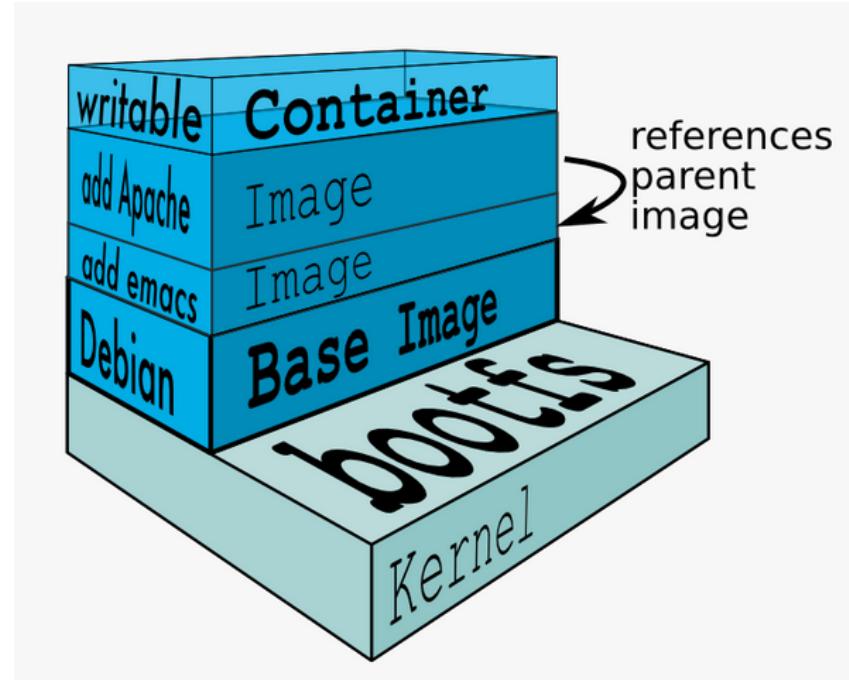
In this module we will

- Explain how image layers work
- Build an image by committing changes in a container
- Learn how to build images with a Dockerfile
- Work through examples of key Dockerfile instructions
 - RUN
 - CMD
 - ENTRYPOINT
 - COPY
- Talk about the best practices when writing a Dockerfile



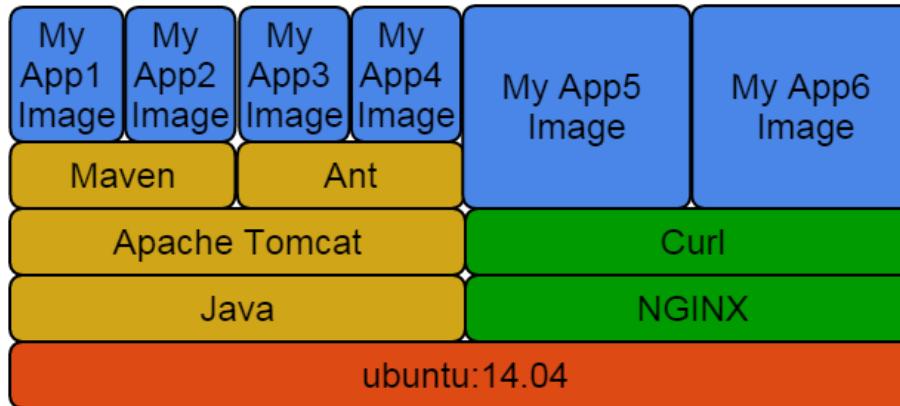
Understanding image layers

- An image is a collection of files and some meta data
- Images are comprised of multiple layers
- A layer is also just another image
- Each image contains software you want to run
- Every image contains a base layer
- Docker uses a copy on write system
- Layers are read only



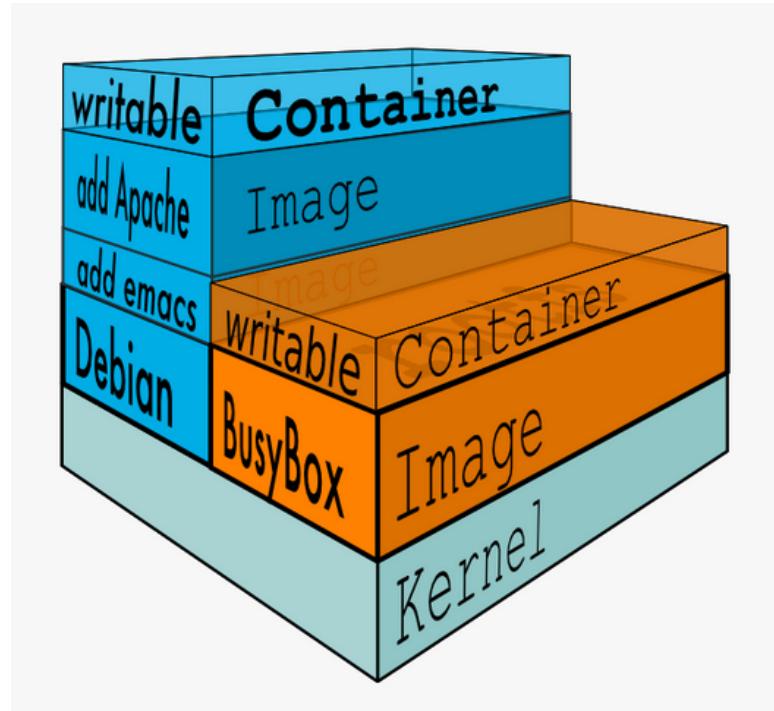
Sharing layers

- Images can share layers in order to speed up transfer times and optimize disk and memory usage
- Parent images that already exists on the host do not have to be downloaded



The container writable layer

- Docker creates a top writable layer for containers
- Parent images are read only
- All changes are made at the writeable layer
- When changing a file from a read only layer, the copy on write system will copy the file into the writable layer



Methods of building images

- Three ways
 - Commit changes from a container as a new image
 - Build from a Dockerfile
 - Import a tarball into Docker as a standalone base layer



Committing changes in a container

- Allows us to build images interactively
- Get terminal access inside a container and install the necessary programs and your application
- Then save the container as a new image using the docker commit command



EX6.1 - Make changes in a container

1. Create a container from an Ubuntu image and run a bash terminal
`docker run -i -t ubuntu:14.04 bash`
2. Run `apt-get update` to refresh the distro packages
3. Install `wget`
4. Install `vim`
`apt-get install -y wget vim`
5. Exit the container terminal



Comparing container changes

- Use the `docker diff` command to compare a container with its parent image
 - Recall that images are read only and changes occur in a new layer
 - The parent image (the original) is being compared with the new layer
- Copy on write system ensures that starting a container from a large image does not result in a large copy operation
- Lists the files and directories that have changed

```
johnnytu@docker-ubuntu:~$ docker diff mad_wilson
C /root
C /root/.bash_history
D /root/test
A /root/test2
```



EX6.2 – Compare changes

1. Run `docker diff` against the container you created in exercise 6.1
2. What can you observe?



Docker Commit

- docker commit command saves changes in a container as a new image
- **Syntax**
docker commit [options] [container ID] [repository:tag]
- Repository name should be based on username/application
- Can reference the container with container name instead of ID

Save the container with ID of 984d25f537c5 as a new image in the repository johnnytu/myapplication. Tag the image as 1.0

```
docker commit 984d25f537c5 johnnytu/myapplication:1.0
```



Image namespaces

- Image repositories belong in one of three namespaces
 - Root
 - ubuntu:14.04
 - centos:7
 - nginx
 - User OR organization
 - johnnytu/myapp
 - mycompany/myapp
 - Self hosted
 - registry.mycompany.com:5000/my-image



Uses for namespaces

- Root namespace is for official repositories
- User and organization namespaces are for images you create and plan to distribute on Docker Hub
- Self-hosted namespace is for images you create and plan to distribute in your own registry server



EX6.3 - Build new image

1. Save your container as a new image. For the repository name use <your name>/myimage. Tag the image as 1.0
`docker commit <container ID> <yourname>/myapp:1.0`
2. Run docker images and verify that you can see your new image
3. Create a container using the new image you created in the previous exercise. Run bash as the process to get terminal access
`docker run -i -t <yourname>/myapp:1.0 bash`
4. Verify that vim and wget are installed
5. Create a file in your container and commit that as a new image. Use the same image name but tag it as 1.1
6. Run docker diff on your container. What do you observe?



Intro to Dockerfile

A **Dockerfile** is a configuration file that contains instructions for building a Docker image

- Provides a more effective way to build images compared to using docker commit
- Easily fits into your development workflow and your continuous integration and deployment process



Process for building images from Dockerfile

1. Create a Dockerfile in a new folder or in your existing application folder
2. Write the instructions for building the image
 - What programs to install
 - What base image to use
 - What command to run
3. Run `docker build` command to build an image from the Dockerfile



Dockerfile Instructions

- Instructions specify what to do when building the image
- **FROM** instruction specifies what the base image should be
- **RUN** instruction specifies a command to execute
- Comments start with “#”

```
#Example of a comment  
FROM ubuntu:14.04  
RUN apt-get install vim  
RUN apt-get install curl
```



FROM instruction

- Must be the first instruction specified in the Dockerfile (not including comments)
- Can be specified multiple times to build multiple images
 - Each FROM marks the beginning of a new image
- Can use any image including, images from official repositories, user images and images in self hosted registries.

Examples

```
FROM ubuntu
```

```
FROM ubuntu:14.04
```

```
FROM johnnytu/myapplication:1.0
```

```
FROM company.registry:5000/myapplication:1.0
```



More about RUN

- RUN will do the following:
 - Execute a command.
 - Record changes made to the filesystem.
 - Works great to install libraries, packages, and various files.
- RUN will NOT do the following:
 - Record state of processes.
 - Automatically start daemons.



EX6.4 – A Simple Dockerfile

1. In your home directory, create a folder called myimage
2. Change directory into the myimage folder and create a new file called Dockerfile
3. Open Dockerfile, write the following and then save the file

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y wget
```



Docker Build

- Syntax

```
docker build [options] [path]
```

- Common option to tag the build

```
docker build -t [repository:tag] [path]
```

Build an image using the current folder as the context path. Put the image in the johnnytu/myimage repository and tag it as 1.0

```
docker build -t johnnytu/myimage:1.0 .
```

As above but use the myproject folder as the context path

```
docker build -t johnnytu/myimage:1.0 myproject
```



EX6.5 – Build the Dockerfile

1. Run `docker build -t myimage .`
2. Observe the output
3. Verify that the image is built by running `docker images`
4. Run a container from your image and verify that `wget` is installed



Build output

```
johnnytu@docker-ubuntu:~/myimage$ docker build -t myimage .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
--> 07f8e8c5e660
Step 1 : RUN apt-get update
--> Running in fd009309d260
...
...
Reading package lists...
--> 161253fe4244
Removing intermediate container fd009309d260
Step 2 : RUN apt-get install -y wget
--> Running in 69ba8a082150
Reading package lists...
...
...
--> c7cc72b567e4
Removing intermediate container 69ba8a082150
Successfully built c7cc72b567e4
```



The build context

```
johnnytu@docker-ubuntu:~/myimage$ docker build -t myimage .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
```

- The build context is the directory that the Docker client sends to the Docker daemon during the `docker build` command
- Directory is sent as an archive
- Docker daemon will build using the files available in the context
- Specifying “.” for the build context means to use the current directory



Examining the build process

- Each RUN instruction will execute the command on the top writable layer of a new container

```
Step 1 : RUN apt-get update  
---> Running in fd009309d260
```

- At the end of the execution of the command, the container is committed as a new image and then deleted

```
---> 161253fe4244  
Removing intermediate container fd009309d260
```



Examining the build process (cont'd)

- The next RUN instruction will be executed in a new container using the newly created image

```
Step 2 : RUN apt-get install -y wget  
---> Running in 69ba8a082150
```

- The container is committed as a new image and then removed. Since this RUN instruction is the final instruction, the image committed is what will be used when we specify the repository name and tag used in the docker build command

```
---> c7cc72b567e4  
Removing intermediate container 69ba8a082150  
Successfully built c7cc72b567e4
```



EX6.6 - The build cache

1. In your myimage folder run the same build command again
`docker build -t myimage .`
2. Notice how the build was much faster. Almost instant.
3. Observe the output. Do you notice anything different?



The build cache

- Docker saves a snapshot of the image after each build step
- Before executing a step, Docker checks to see if it has already run that build sequence previously
- If yes, Docker will use the result of that instead of executing the instruction again
- Docker uses exact strings in your Dockerfile to compare with the cache
 - Simply changing the order of instructions will invalidate the cache
- To disable the cache manually use the `--no-cache` flag
`docker build --no-cache -t myimage .`



EX6.7 – Experiment with the cache

1. Open your Dockerfile and add another RUN instruction to install vim
2. Build the image and observe the output. Notice the cache being used for the RUN instructions we had defined previously but not for the new one
3. Build the image again and notice the cache being used for all three RUN instructions
4. Edit the Dockerfile and swap the order of the two apt-get install commands
5. Build the image again and notice the cache being invalidated for both apt-get install instructions



Multiple commands in a single RUN Instruction

- Use the shell syntax “`&&`” to combine multiple commands in a single RUN instruction
- Commands will all be run in the same container and committed as a new image at the end
- Reduces the number of image layers that are produced

```
RUN apt-get update && apt-get install -y \
    curl \
    vim \
    openjdk-7-jdk
```



Viewing Image layers and history

- docker history command shows us the layers that make up an image
- See when each layer was created, its size and the command that was run

IMAGE	CREATED	CREATED BY	SIZE
10f1e1747aa1	12 seconds ago	/bin/sh -c apt-get install -y wget	6.119 MB
9b6aeeef1e9cc	23 seconds ago	/bin/sh -c apt-get install -y vim	43.12 MB
334d0289feff	10 minutes ago	/bin/sh -c apt-get update	20.86 MB
07f8e8c5e660	2 weeks ago	/bin/sh -c #(nop) CMD ["/bin/bash"]	0 B
37bea4ee0c81	2 weeks ago	/bin/sh -c sed -i 's/^#\s*/(deb.*universe)\\$/	1.895 kB
a82efea989f9	2 weeks ago	/bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic	194.5 kB
e9e06b06e14c	2 weeks ago	/bin/sh -c #(nop) ADD file:f4d7b4b3402b5c53f2	188.1 MB



EX6.8 (optional) – Build history

1. Examine the build history of the image built from the Dockerfile
2. Note down the image id of the layer with the `apt-get update` command
3. Now aggregate the two RUN instructions with the `apt-get install` command

```
RUN apt-get install -y wget vim
```

4. Build the image again
5. Run `docker history` on the image. What do you observe about the image layers



CMD Instruction

- CMD defines a default command to execute when a container is created
- Shell format and EXEC format
- Can only be specified once in a Dockerfile
 - If specified multiple times, the last CMD instruction is executed
- **Can be overridden at run time**

Shell format

```
CMD ping 127.0.0.1 -c 30
```

Exec format

```
CMD ["ping", "127.0.0.1", "-c", "30"]
```



EX6.9 - Try CMD

1. Go into the `myimage` folder and open your Dockerfile from the previous exercise

2. Add the following line to the end

```
CMD ["ping", "127.0.0.1", "-c", "30"]
```

3. Build the image

```
docker build -t <yourname>/myimage:1.0 .
```

4. Execute a container from the image and observe the output

```
docker run <yourname>/myimage:1.0
```

5. Execute another container from the image and specify the echo command

```
docker run <yourname>/myimage:1.0 echo "hello world"
```

6. Observe how the container argument overrides the CMD instruction



ENTRYPOINT Instruction

- Defines the command that will run when a container is executed
- Run time arguments and `CMD` instruction are passed as parameters to the `ENTRYPOINT` instruction
- Shell and EXEC form
- Container essentially runs as an executable

```
ENTRYPOINT ["ping"]
```



EX6.10 - Entrypoint

1. Open your Dockerfile in the `myimage` folder
2. Delete the `CMD` instruction line and replace it with
`ENTRYPOINT ["ping"]`
3. Build the image
4. Run a container from the image but do not specify any commands or arguments
`docker run myimage:1.0`
5. What do you notice?
6. Now run a container and specify an argument of `127.0.0.1`
`docker run myimage:1.0 127.0.0.1`



Using CMD with ENTRYPOINT

- If ENTRYPOINT is used, the CMD instruction can be used to specify default parameters
- Parameters specified during docker run will override CMD
- If no parameters are specified during docker run, the CMD arguments will be used for the ENTRYPOINT command



EX6.11 – CMD and ENTRYPOINT

1. Open your Dockerfile and modify the ENTRYPOINT instruction to include 2 arguments for the ping command
ENTRYPOINT ["ping", "-c", "50"]
2. Add a CMD instruction and specify the IP 127.0.0.1
CMD ["127.0.0.1"]
3. Save the file and build the image
4. Run a container and specify your IP address as the parameter
docker run myimage:1.0 <your ip>
5. Now run another container but do not specify any parameters. What do you observe?
docker run myimage:1.0



Shell vs exec format

- The RUN, CMD and ENTRYPOINT instructions can be specified in either shell or exec form

In shell form, the command will run inside a shell with /bin/sh -c

```
RUN apt-get update
```

Exec format allows execution of command in images that don't have /bin/sh

```
RUN ["apt-get", "update"]
```



Shell vs exec format

- Shell form is easier to write and you can perform shell parsing of variables
- For example

```
CMD sudo -u ${USER} java ....
```
- Exec form does not require image to have a shell
- For the ENTRYPOINT instruction, using shell form will prevent the ability to specify arguments at run time
 - The CMD arguments will not be used as parameters for ENTRYPOINT



Overriding ENTRYPOINT

- Specifying parameters during `docker run` will result in your parameters being used as arguments for the `ENTRYPOINT` command
- To override the command specified by `ENTRYPOINT`, use the `--entrypoint` flag.
- Useful for troubleshooting your images

Run a container using the image “myimage” and specify to run a bash terminal instead of the program specified in the image `ENTRYPOINT` instruction

```
docker run -it --entrypoint bash myimage
```



Copying source files

- When building “real” images you would want to do more than just install some programs
- Examples
 - Compile your source code and run your application
 - Copy configuration files
 - Copy other content
- How do we get our content on our host into the container?
- Use the COPY instruction



COPY instruction

- The `COPY` instruction copies new files or directories from a specified **source** and adds them to the container filesystem at a specified **destination**
- **Syntax**
`COPY <src> <dest>`
- The `<src>` path must be inside the build context
- If the `<src>` path is a directory, all files in the directory are copied. The directory itself is not copied
- You can specify multiple `<src>` directories



COPY examples

Copy the server.conf file in the build context into the root folder of the container

```
COPY server.conf /
```

Copy the files inside the data/server folder of the build context into the /data/server folder of the container

```
COPY data/server /data/server
```



Dockerize an application

- The Dockerfile is essential if we want to adapt our existing application to run on containers
- Take a simple Java program as an example. To build and run it, we need the following on our host
 - The Java Development Kit (JDK)
 - The Java Virtual Machine (JVM)
 - Third party libraries depending on the application itself
- You compile the code, run the application and everything looks good



Dockerize an application

- Then you distribute the application and run it on a different environment and it fails
- Reasons why the Java application fails?
 - Missing libraries in the environment
 - Missing the JDK or JVM
 - Wrong version of libraries
 - Wrong version of JDK or JVM
- So why not run your application in a Docker container?
- Install all the necessary libraries in the container
- Build and run the application inside the container and distribute the image for the container
- Will run on any environment with the Docker Engine installed



EX6.12 – Setup sample application

1. Install java 7

```
sudo apt-get install openjdk-7-jdk
```

2. In your home directory, create a folder called javahelloworld
3. In your new folder, create a file called HelloWorld.java
4. Write the following code

```
public class HelloWorld
{
    public static void main (String [] args)
    {
        System.out.println("hello world");
    }
}
```



EX6.12 – Setup sample application

5. Compile your HelloWorld.java class

```
javac HelloWorld.java
```

6. Run the program

```
java HelloWorld
```



EX6.13 – Dockerize the application

1. In the `javadelloworld` folder, create a Dockerfile
2. Use `java:7` as your base image. This image contains the JDK needed to build and run our application

```
FROM java:7
```

3. Copy your source file into the container root folder

```
COPY HelloWorld.java /
```

4. Add an instruction to compile your code

```
RUN javac HelloWorld.java
```

5. Add an instruction to run your program when running the container

```
ENTRYPOINT ["java", "HelloWorld"]
```

6. Build the image

7. Run a container from the image and observe the output



EX6.13 - Dockerize the application

8. Let's take a look inside the container. Run another container and override the ENTRYPOINT instruction. Specify to run a bash terminal
`docker run -it --entrypoint bash <your java image>`
9. Find where your HelloWorld.java source file is
10. Find where the compiled code is



Specify a working directory

- Previously all our instructions have been executed at the root folder in our container
- `WORKDIR` instruction allows us to set the working directory for any subsequent `RUN`, `CMD`, `ENTRYPOINT` and `COPY` instructions to be executed in
- Syntax
`WORKDIR /path/to/folder`
- Path can be absolute or relative to the current working directory
- Instruction can be used multiple times



EX6.14 – Restructure our application

1. In your `javahelloworld` folder, create two folders called `src` and `bin`
2. Move your `HelloWorld.java` source file into `src`
3. Compile your code and place the compiled class into the `bin` folder
`javac -d bin src/HelloWorld.java`
4. Run your application
`java -cp bin HelloWorld`



EX6.15 – Redefine our Dockerfile

1. Open your Dockerfile
2. Modify the COPY instruction to copy all files in the src folder into /home/root/javahelloworld/src
COPY src /home/root/javahelloworld/src
3. Modify the RUN instruction to compile the code by referencing the correct src folder and to place the compiled code into the bin folder
javac -d bin src/HelloWorld.java
4. Modify ENTRYPOINT to specify java -cp bin
ENTRYPOINT ["java", "-cp", "bin", "HelloWorld"]
5. Build your image. Notice the error?



EX6.16 – Specify the working directory

1. Before the `RUN` instruction, add a `WORKDIR` instruction to specify where to execute your commands
`WORKDIR /home/root/javahelloworld`
2. Build your image. Notice there is still an error.
3. Add another `RUN` instruction after `WORKDIR` to create the `bin` folder
`RUN mkdir bin`
4. Build your image.
5. Run a container from your image and verify you can see the “hello world” output
6. Override the `ENTRYPOINT` and run `bash`. Check your folder structure and verify you have `src` and `bin` inside `/home/root/javahelloworld`
`docker run -it --entrypoint bash <your image>`



MAINTAINER Instruction

- Specifies who wrote the Dockerfile
- Optional but best practice to include
- Usually placed straight after the FROM instruction

Example

```
MAINTAINER Docker Training <education@docker.com>
```



ENV instruction

- Used to set environment variables in any container launched from the image
- Syntax

```
ENV <variable> <value>
```

Examples

```
ENV JAVA_HOME /usr/bin/java
```

```
ENV APP_PORT 8080
```



EX6.17 – Environment variables

1. Modify the `javadelloworld` image Dockerfile and set a variable called “`foo`”. Specify any value

```
ENV FOO bar
```

2. Build the image
3. Run a container from the image and check for the presence of the variable



ADD instruction

- The ADD instruction copies new files or directories from a specified **source** and adds them to the container filesystem at a specified **destination**
- Syntax
ADD <src> <dest>
- The <src> path is relative to the directory containing the Dockerfile
- If the <src> path is a directory, all files in the directory are copied. The directory itself is not copied
- You can specify multiple <src> directories

Example

```
ADD /src /myapp/src
```



COPY vs ADD

- Both instructions perform a near identical function
- ADD has the ability to auto unpack tar files
- ADD instruction also allows you to specify a URL for your content (although this is not recommended)
- Both instructions use a checksum against the files added. If the checksum is not equal then the test fails and the build cache will be invalidated
 - Because it means we have modified the files



Best practices for writing Dockerfiles

- Remember, each line in a Dockerfile creates a new layer if it changes the state of the image
- You need to find the right balance between having lots of layers created for the image and readability of the Dockerfile
- Don't install unnecessary packages
- One ENTRYPOINT per Dockerfile
- Combine similar commands into one by using “`&&`” and “`\`”

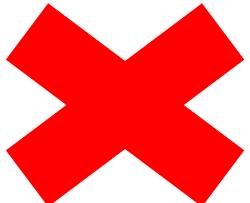
Example

```
RUN apt-get update && \
    apt-get install -y vim && \
    apt-get install -y curl
```



Best practices for writing Dockerfiles

- Use the caching system to your advantage
 - The order of statements is important
 - Add files that are least likely to change first and the ones most likely to change last
- The example below is not ideal because our build system does not know whether the requirements.txt file has changed



```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -q \
    python-all python-pip
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
RUN pip install -qr requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```



Best practices for writing Dockerfiles

- **Correct way** would be in the example below.
- `requirements.txt` is added in a separate step so Docker can cache more efficiently. If there's no change in the file the RUN pip install instruction does not have to execute and Docker can use the cache for that layer.
- The rest of the files are added afterwards



```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y -q \
    python-all python-pip
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
CMD ["python", "app.py"]
```



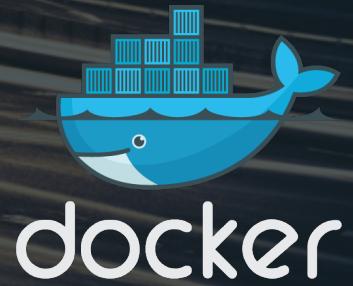
Module summary

- Images are made up of multiple layers
- Committing changes we make in a container as a new image is a simple way to create our own images but is not a very effective method as part of a development workflow
- A Dockerfile is the preferred method of creating images
- Key Dockerfile instructions we learnt about
 - RUN
 - CMD
 - ENTRYPOINT
 - COPY
- Key commands
 - docker build



Module 7:

Managing and Distributing Images



Module objectives

In this module we will

- Outline where we can distribute our images to
- Push an image into Docker Hub
- Setup public and private repositories in Docker Hub
- Explain how image tagging works
- Learn how to delete images



Distributing your image

- To distribute your image there are two options
 - Push to Docker Hub
 - Push to your own registry server
 - `docker export` and `docker import` commands
- Images in Docker Hub can reside in public or private repositories
- Registry server can be setup to be publicly available or behind the firewall



Docker Hub Repositories

- Users can create their own repositories on Docker Hub
- Public and Private
- Push local images to a repository

The screenshot shows the Docker Hub interface for the 'trainingteam' organization. At the top, there's a navigation bar with links for Dashboard, Explore, and Organizations. A search bar contains the query 'java'. On the right, there's a user icon and the text 'trainingteam'. Below the navigation, there are filters for 'Repositories', 'Stars', and 'Contributed'. A message indicates 'Private Repositories: Using 1 of 1' with a link to 'Get more'. The main area is titled 'Repositories' and contains a search bar. Two repository cards are listed:

- trainingteam/testexample** (public)
 - 0 STARS
 - 16 PULLS
 - [DETAILS](#)
- trainingteam/helloworld** (public)
 - 0 STARS
 - 12 PULLS
 - [DETAILS](#)

A large red box highlights the 'Create Repository +' button in the top right corner of the main content area. To the right of the repositories, there's a sidebar for 'Docker Trusted Registry' with the text: 'Need an on-premise registry? Get a 30-day free trial'.



Creating a repository

- Repository will reside in the user or organization namespace. For example:
 - trainingteam/myrepo
 - johnny/myrepo
- Public repositories are listed and searchable for public use
- Anyone can pull images from a public repository



Creating a repository

trainingteam ▾ javaHelloWorld

A simple hello world

Hello World program written in Java and running in a Docker container|

Visibility

public ▾

Create



Repository description

- Once a repository has been created we can write a detailed description about the images
- Good to include instructions on how to run the images
- You may want to
 - Link to the source repository of the application the image is designed to run
 - Link to the source of the Dockerfile
- Description is written in markdown
<http://daringfireball.net/projects/markdown/syntax>



EX7.1 – Create a public repository

1. Login to your Docker Hub account
2. Create a new public repository called `javadelloworld`
3. Write a basic description for your repository



Pushing Images to Docker Hub

- Use `docker push` command
- Syntax
`docker push [repo:tag]`
- Local repo must have same name and tag as the Docker Hub repo
- Only the image layers that have changed get pushed
- You will be prompted to login to your Docker Hub account
- **Note:** You don't need to create the repository on Docker Hub first.
 - If you push a local repository that does not exist on Docker Hub, it will be automatically created



Pushing Images

```
johnnytu@docker-ubuntu:~/javahelloworld$ docker push trainingteam/javahelloworld:1.0
The push refers to a repository [trainingteam/javahelloworld] (len: 1)
b8a9f23d0df8: Image push failed

Please login prior to push:
Username: trainingteam
Password:
Email: training@servicerocket.com
WARNING: login credentials saved in /home/johnnytu/.dockercfg.
Login Succeeded
The push refers to a repository [trainingteam/javahelloworld] (len: 1)
b8a9f23d0df8: Image already exists
c9b2cded3b61: Image successfully pushed
0b94e15ddfae: Image successfully pushed
4342503c37c5: Image successfully pushed
c7e746b8760e: Image successfully pushed
31dd6207396b: Image successfully pushed
760f8f0deb51: Image successfully pushed
91298d5a4caf: Image successfully pushed
22f522207fc7: Image successfully pushed
bf9f6b703af2: Image successfully pushed
05bacbdafa6eb: Image successfully pushed
e66a33f451f4: Image successfully pushed
41b730702607: Image successfully pushed
3cb35ae859e7: Image successfully pushed
Digest: sha256:9b9ae810e844b14182ceda74b06c7d9a7fa21513c76a08fd8e66798416b150fc
```



EX7.2 – Push image to Docker Hub

1. Try to push the javahelloworld image we created in Exercise 6.16 to Docker Hub

```
docker push javahelloworld
```

2. Notice the error message saying that you cannot push a root repository



Tagging Images

- Used to rename a local image repository before pushing to Docker Hub
- Syntax:

```
docker tag [image ID] [repo:tag]
```

OR

```
docker tag [local repo:tag] [Docker Hub repo:tag]
```

Tag image with ID (trainingteam/testexample is the name of repository on Docker hub)

```
docker tag edfc212de17b trainingteam/testexample:1.0
```

Tag image using the local repository tag

```
docker tag johnnytu/testimage:1.5 trainingteam/testexample
```



One image, many tags

- The same image can have multiple tags
- Image can be identified by it's ID
 - The ID is generated using a hash of the image content for consistency

REPOSITORY SIZE	TAG	IMAGE ID	CREATED	VIRTUAL
trainingteam/javahelloworld	1.1	76b3b2455967	5 minutes ago	598.1 MB
trainingteam/testimage	1.0	ee8800b0677b	8 minutes ago	263.8 MB
javahelloworld	1.0	b8a9f23d0df8	3 hours ago	588.7 MB
javahelloworld	latest	b8a9f23d0df8	3 hours ago	588.7 MB
trainingteam/javahelloworld	1.0	b8a9f23d0df8	3 hours ago	588.7 MB
java	7	31dd6207396b	2 weeks ago	588.7 MB
ubuntu	14.04	07f8e8c5e660	2 weeks ago	188.3 MB



Tagging images for a push

Local repo name

```
johnnytu@docker-ubuntu:~/javahelloworld$ docker images
REPOSITORY          TAG        IMAGE ID      CREATED       VIRTUAL SIZE
javahelloworld     1.0        b8a9f23d0df8   2 hours ago  588.7 MB
java                7          31dd6207396b   2 weeks ago  588.7 MB
johnnytu@docker-ubuntu:~/javahelloworld$
```

Repo name on Docker Hub



EX7.3 – Tag and push image

1. Tag your local image so that it has the same repository name as the Docker Hub repository created in exercise 7.1

```
docker tag javahelloworld:1.0 <username>/javahelloworld:1.0
```

2. List your images and verify the image you just tagged

3. Now try and push the image

```
docker push <username>/javahelloworld:1.0
```

4. Navigate to your repository on Docker Hub and open the Tags tab

5. Verify that you can see the tag 1.0



EX7.4 – Modify image and push

1. Go into the `javadelloworld` folder and edit the Dockerfile
2. Add another `RUN` instruction just before `ENTRYPOINT` and install an application of your choice.
3. Save and build the image. Tag the image as `1.1`
`docker build -t trainingteam/javadelloworld:1.1`
4. Push the image to Docker Hub
5. Observe the output. What do you notice?



Private repositories

- Docker Hub private repositories are not available to the public and will not be listed in search results
- Only you can push and pull images to your repository
- Additional users who want access need to be added as collaborators



EX7.5 - Create a private repository

1. Create a private repository in Docker Hub called myapplication
2. Push the image from exercise 6.5 (repo name myimage) into the repository. Remember, you will have to tag the image first.
docker tag myimage:1.0 <username>/myapplication:1.0
docker push <username>/myapplication:1.0
3. Collaborate with another student in the class and try to download their image from their private repository.
4. Notice how it doesn't work



Collaborators

- Collaborators can push and pull from a private repository but do not have administrative access
- Collaborators will be able to see the private repository on their “Repositories” page
- You also need to be a collaborator if you want to push to a public repository

PUBLIC REPOSITORY

[trainingteam/javahelloworld](#) ☆

Last pushed: never

[Repo Info](#) [Tags](#) [Description](#) [Collaborators](#) [Webhooks](#) [Delete Repository](#)

Collaborators

Username	Access	Action



Adding collaborators

- Click on the “collaborators” tab on the top navigation bar
- Specify the Docker Hub username of the person you wish to add

The screenshot shows the Docker Hub repository settings interface. The top navigation bar includes tabs for Repo Info, Tags, Description, Collaborators (which is highlighted in blue), Webhooks, and Delete Repository. Below the navigation bar, the title "Collaborators" is displayed. A table lists a single collaborator: "johnnytu" with "Collaborator" access. To the right of the table is a modal window titled "Add User" containing a "Username" field with "johnnytu" typed into it and a "Add User" button.

Username	Access	Action
johnnytu	Collaborator	<button>✖ Remove</button> Success



EX7.6 – Add collaborators

1. Add another student in the class as a collaborator in your private repository
2. Get that person to verify that they can see your private repository on their “Repositories” page
3. Get that person to download an image from your private repository



Deleting local Images

- Use `docker rmi` command
- `docker rmi [image ID]`
or
`docker rmi [repo:tag]`
- If an image is tagged multiple times, remove each tag

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
test1	latest	cbfa5ab76a11	12 seconds ago	262.5 MB
test	latest	cbfa5ab76a11	12 seconds ago	262.5 MB


```
johnnytu@dockertraining:~/test$ docker rmi test
Untagged: test:latest
johnnytu@dockertraining:~/test$ docker rmi test1
Untagged: test1:latest
Deleted: cbfa5ab76a11eec84b751ae261d3f870a0be61bb899e651c857ae4cc3eed9bc9
```



EX7.7 – Delete images

1. Use the `docker rmi` command and delete the image you downloaded from the other student's private repository

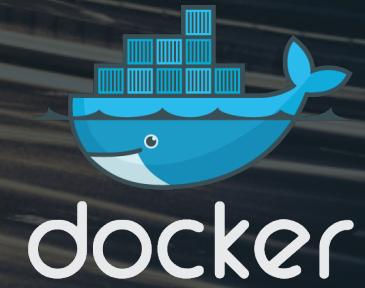


Module summary

- Image can be pushed to the public registry (Docker Hub) or your own private Registry
- To push an image to a Docker Hub repository, the local repository name and tag must be the same
- Image deletion is based on the image tags
- Key commands
 - docker push
 - docker tag



Module 8: Volumes



Module objectives

In this module we will

- Explain what volumes are and what they are used for
- Learn the different methods of mounting a volume in a container
- Mount volumes during the `docker run` command and also in a Dockerfile
- Explain how data containers work
- Create some data containers



Volumes

A **Volume** is a designated directory in a container, which is designed to persist data, independent of the container's life cycle

- Volume changes are excluded when updating an image
- Persist when a container is deleted
- Can be mapped to a host folder
- Can be shared between containers



Volumes and copy on write

- Volumes bypass the copy on write system
- Act as passthroughs to the host filesystem
- When you commit a container as a new image, the content of the volumes will not be brought into that image
- If a `RUN` instruction in a Dockerfile changes the content of a volume, those changes are not recorded either.



Uses of volumes

- De-couple the data that is stored, from the container which created the data
- Good for sharing data between containers
 - Can setup a data containers which has a volume you mount in other containers
 - Share directories between multiple containers
- Bypassing the copy on write system to achieve native disk I/O performance
- Share a host directory with a container
- Share a single file between the host and container



Docker volume command

- The `docker volume` command contains a number of sub commands used to create and manage volumes
- Commands are
 - `docker volume create`
 - `docker volume ls`
 - `docker volume inspect`
 - `docker volume rm`



Creating a volume

- Use the docker volume create command and specify the --name option
- Specify a name so you can easily find and identify your volume later

Create a volume named test1

```
docker volume create --name test1
```



Listing volumes

- Use `docker volume ls` command to display a list of all volumes
- This includes volumes that were mounted to containers using the old `docker run -v` method
- Volumes that were not given a name during creation will have a randomly generated name

```
student@dockerhost:~$ docker volume ls
DRIVER      VOLUME NAME
local      51ef54f83ccb4d7990ed95aebd78bb130a3c9db09fac1b3563c80328c737df47
local      7570cec94594dc044176f27877f9b2548fc1399b85b6c2304a27d09d133aa11a
local      shared-data
local      test1
local      myvol2
```



Mount a Volume

- Volumes can be mounted when running a container
- Use the `-v` option on `docker run` command and specify the name of the volume and the mount path
syntax: `docker run -v <name>:<path> ...`
- Path is the container folder where you want to mount the volume
- Can mount multiple volumes by using the `-v` option multiple times

Execute a new container and mount the volume test1 in the folder /www/test1

```
docker run -it -v test1:/www/test1 ubuntu:14.04 bash
```

Example of mounting multiple volumes

```
docker run -d -v test1:/www/test1 -v test2:/www/test2 nginx
```



EX8.1 - Create and mount a Volume

1. Create a volume called test1

```
docker volume create --name test1
```

2. Run docker volume ls and verify that you can see your test1 volume

3. Execute a new Ubuntu container and mount the test1 volume. Map it to the path /www/website and run bash as your process

```
docker run -it -v test1:/www/website ubuntu:14.04 bash
```

4. Inside the container, verify that you can get to /www/website
cd /www/website

5. Create a file called test.txt inside the /www/website folder
touch test.txt

6. Exit the container without stopping it by hitting CTRL + P + Q



EX8.1 – (cont'd)

7. Commit the updated container as a new image called test and tag it as 1.0

```
docker commit <container ID> test:1.0
```

8. Execute a new container with your test image and go into its bash shell

```
docker run -it test:1.0 bash
```

9. Verify that the /www/website folder exists and that there are no files inside

10. Exit the container

```
exit
```

11. Run docker ps to ensure that your first container is still running



Where are our volumes?

- Volumes exist independently from containers
- If a container is stopped, we can still access our volume
- To find where the volume is use `docker inspect` on the container and look for the “source” field as shown below

```
"Mounts": [
    {
        "Name": "test1",
        "Source": "/var/lib/docker/volumes/test1/_data",
        "Destination": "/www/website",
        "Driver": "local",
        "Mode": "z",
        "RW": true
    }
],
```



Docker volume inspect command

- The docker volume inspect command shows all the information about a specified volume
- Information includes the “Mountpoint” which tells us where the volume is located on the host

```
student@dockerhost:~$ docker volume inspect test1
[
  {
    "Name": "test1",
    "Driver": "local",
    "Mountpoint": "/var/lib/docker/volumes/test1/_data"
  }
]
```



EX8.2 – Find your volume

1. Run `docker volume inspect` on the `test1` volume
`docker volume inspect test1`
2. Copy the path specified by the `Mountpoint` field. The path should be
`/var/lib/docker/volumes/test1/_data`
3. Elevate your user privileges to root
`sudo su`
4. Change directory into the volume path in step 2
`cd /var/lib/docker/volumes/test1/_data`
5. Run `ls` and verify you can see the `test.txt` file



EX8.2 – (cont'd)

6. Create another file called test2.txt

```
touch test2.txt
```

7. Exit the superuser account

```
exit
```

8. Use docker exec to log back into the shell of your Ubuntu container that is still running

```
docker exec -it <container name> bash
```

9. Change directory into the /www/website folder

10. Verify that you can see both the test.txt and test2.txt files



Deleting a volume

- Volumes are not deleted when you delete a container
- Use the `docker volume rm` command to delete a volume
- Can also use the `-v` option in the `docker rm` command to delete all the volumes associated with the container when deleting the container itself

Delete the volume called test1

```
docker volume rm test1
```

Delete a container and remove it's associated volumes

```
docker rm -v <container ID>
```



Deleting volumes

- You cannot delete a volume if it is being used by a container
 - Doesn't matter if the container is running or stopped
- Must delete all containers first
- `docker rm -v` command will not delete a volume associated with the container if that volume is mounted in another container



EX8.3 – Deleting volumes

1. Delete the container from exercise 8.1 without using any options

```
docker rm <container ID>
```

2. Run docker volume ls and check the result

3. Notice our test1 volume is still present

4. Elevate your user privileges

```
sudo su
```

5. Change directory to the volume path and check to see that the test.txt and test2.txt files are still present

```
cd /var/lib/docker/volumes/test1/_data  
ls
```

6. Exit superuser

```
exit
```

7. Delete the test1 volume

```
docker volume rm test1
```

8. Run docker volume ls and make sure the test1 volume is no longer displayed



Mounting host folders to a volume

- When running a container, you can map folders on the host to a volume
- The files from the host folder will be present in the volume
- Changes made on the host are reflected inside the container volume
- Syntax

```
docker run -v [host path]:[container path]:[rw|ro]
```

- rw or ro controls the write status of the volume



Simple Example

- In the example below, files inside `/home/user/public_html` on the hosts will appear in the `/data/www` folder of the container
- If the host path or container path does not exist, it will be created
- If the container path is a folder with existing content, the files will be replaced by those from the host path

Mount the contents of the `public_html` folder on the hosts to the container volume at `/data/www`

```
docker run -d -v /home/user/public_html:/data/www ubuntu
```



Inspecting the mapped volume

- The `Mounts` field from `docker inspect` will show the container volume being mapped to the host path specified during `docker run`

```
"Mounts": [  
  {  
    "Source": "/home/student/public_html",  
    "Destination": "/data/www",  
    "Mode": "",  
    "RW": true  
  },  
]
```



EX8.4 – Mount a host folder

1. In your home directory, create a `public_html` folder and create an `index.html` file inside this folder
2. Run an Ubuntu container and mount the `public_html` folder to the volume `/data/www`
`docker run -it -v /home/<user>/public_html:/data/www`
`ubuntu:14.04`
3. In the container, look inside the `/data/www` folder and verify you can see the `index.html` file
4. Exit the container without stopping it
`CTRL + P + Q`
5. Modify the `index.html` file on the host by adding some new lines
6. Attach to the container and check the `index.html` file inside `/data/www` for the changes

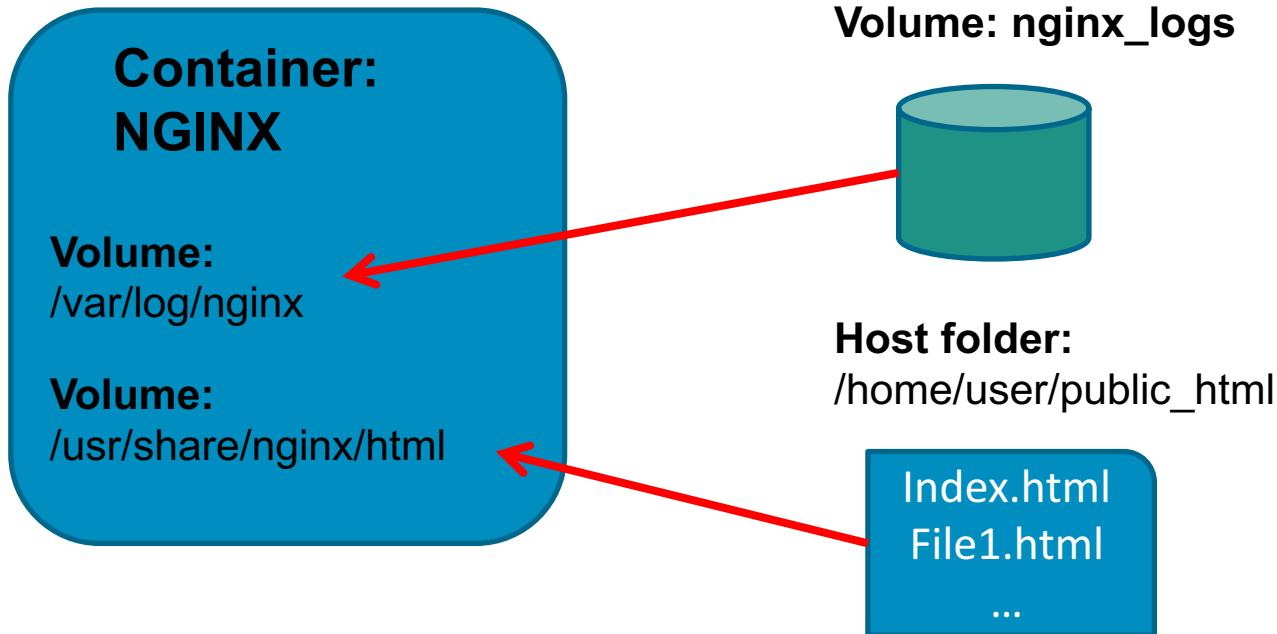


A data persistence example with NGINX

- Let's run an NGINX container and have it serve web pages that we have on the host machine
- That way we can conveniently edit the page on the host instead of having to make changes inside the container
- Mount a volume to the NGINX folder where logs are written to. This way we can persist the log files
- Quick NGINX 101
 - NGINX starts with a one default server on port 80
 - Default location for pages is the `/usr/share/nginx/html` folder
 - By default the folder has an `index.html` file which is the welcome page
 - Log files are written to the `/var/log/nginx` folder



Diagram of example



The NGINX welcome page

`index.html` page in
folder
`/usr/share/nginx/html`

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.



Running the container

- Key aspects to remember when running the container
 - Use `-d` to run it in detached mode
 - Use `-P` to automatically map the container ports (more on this in Module 9)
- When we visit our server URL we should now see our `index.html` file inside our `public_html` folder instead of the default NGINX welcome page

Run an nginx container and map the our public_html folder to the volume at the /usr/share/nginx/html folder in the container. <path> is the path to public_html

```
docker run -d -P -v <path>:/usr/share/nginx/html nginx
```



Custom container names

- By default, containers we create, have a randomly generated name
- To give your container a specific name, use the `--name` option on the `docker run` command
- Existing container can be renamed using the `docker rename` command
`docker rename <old name> <new name>`

Create a container and name it mynginx

```
docker run -d -P --name mynginx nginx
```

Rename the container called `happy_einstein` to `mycontainer`

```
docker rename happy_einstein mycontainer
```



EX8.5 – Run NGINX container

1. Create a volume called nginx_logs

```
docker volume create --name nginx_logs
```

2. Run an NGINX container and map your public_html folder to a volume at /usr/share/nginx/html. Also mount your nginx_logs volume to the /var/log/nginx folder. Name the container nginx_server

```
docker run -d -P --name nginx_server \
-v ~/public_html:/usr/share/nginx/html \
-v nginx_logs:/var/log/nginx \
nginx
```

3. Get terminal access to your container

```
docker exec -it nginx_server bash
```



EX8.5 – (cont'd)

4. Check the `/usr/share/nginx/html` folder for your `index.html` file and then exit the terminal
5. Run `docker ps` to find the host port which is mapped to port 80 on the container
6. On your browser, access your AWS server URL and specify the port from question 5)
7. Verify you can see the contents of your `index.html` file from your `public_html` folder
8. Now modify your `index.html` file
9. Refresh your browser and verify that you can see the changes



EX8.6 – Check log persistence

1. Get terminal access to your container again
`docker exec -it nginx_server bash`
2. Change directory to /var/log/nginx
`cd /var/log/nginx`
3. Check that you can see the `access.log` and `error.log` files
4. Run `tail -f access.log`, refresh your browser a few time and observe the log entries being written to the file
5. Exit the container terminal
6. Run `docker volume inspect nginx_logs` and copy the path indicated by the Mountpoint field (Path should be `/var/lib/docker/volumes/nginx_logs/_data`)



EX8.6 – (cont'd)

7. Elevate user privileges

```
sudo su
```

8. Change directory into the volume path

```
cd /var/lib/docker/volumes/nginx_logs/_data
```

9. Check for the presence of the `access.log` and `error.log` file

10. Run `tail -f access.log`, refresh your browser a few times in order to make some requests to the NGINX server

11. Observe log entries being written into the `access.log` file



Use cases for mounting host directories

- You want to manage storage and snapshots yourself.
 - With LVM, or a SAN, or ZFS, or anything else!
- You have a separate disk with better performance (SSD) or resiliency (EBS) than the system disk, and you want to put important data on that disk.
- You want to share your source directory between your host (where the source gets edited) and the container (where it is compiled or executed).
 - Good for testing purposes but not for production deployment

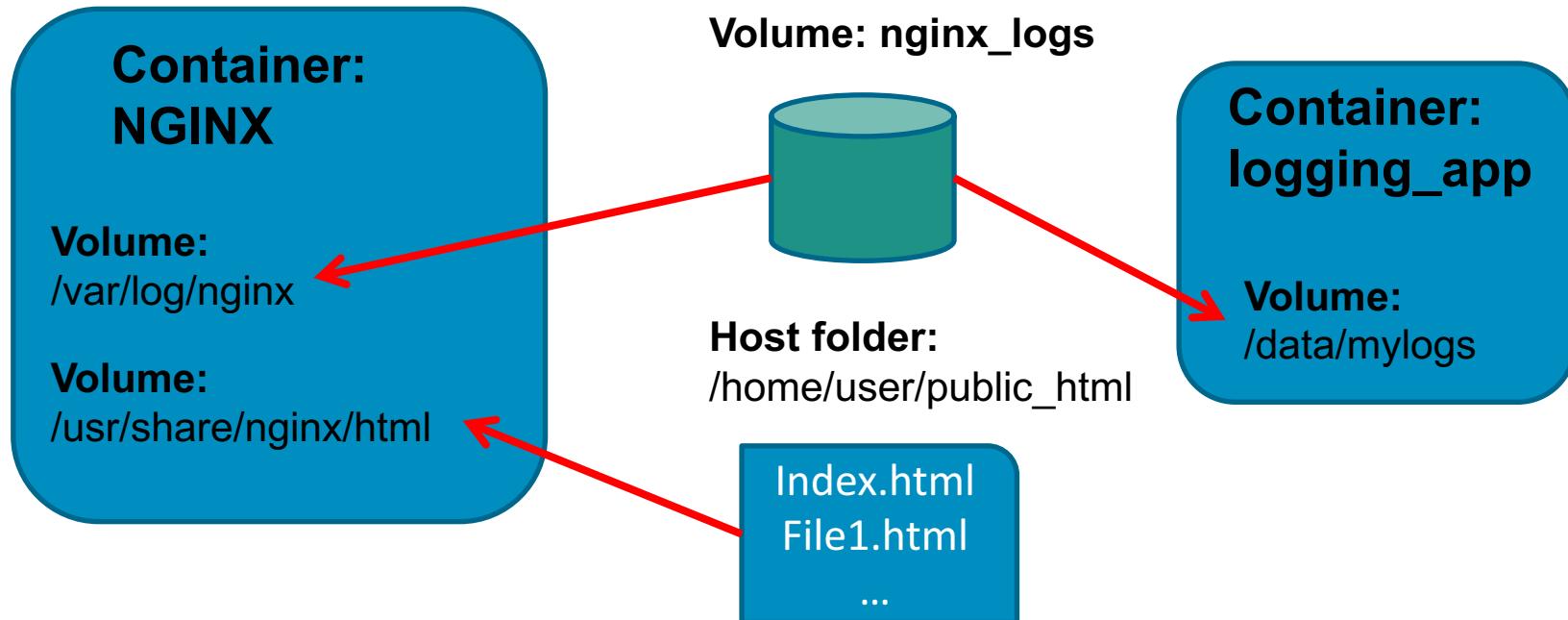


Sharing data between containers

- Volumes can be mounted into multiple containers
- Allows data to be shared between containers
- Example use cases
 - One container writes metric data to the volume
 - Another container runs an application to read the data and generate graphs
- **Note:** Be aware of potential conflicts if multiple applications are allowed write access into the same volume



Extending our NGINX example



EX8.7 – Sharing volumes

1. Run `docker ps` and make sure that your `nginx_server` container from EX8.6 is still running
2. Run an Ubuntu container and mount the `nginx_logs` volume to the folder `/data/mylogs` as read only. Run `bash` as your process.

```
docker run -it \
-v nginx_logs:/data/mylogs:ro
ubuntu:14.04 bash
```
3. On your container terminal, change directory to `/data/mylogs`
4. Confirm that you can see the `access.log` and `error.log` files
5. Try and create a new file called `text.txt`
`touch test.txt`
6. Notice how it fails because we mounted the volume as read only



Volumes in Dockerfile

- VOLUME instruction creates a mount point
- Can specify arguments in a JSON array or string
- Cannot map volumes to host directories
- Volumes are initialized when the container is executed

String example

```
VOLUME /myvol
```

String example with multiple volumes

```
VOLUME /www/websitel.com /www/website2.com
```

JSON example

```
VOLUME ["myvol", "myvol2"]
```



Example Dockerfile with Volumes

- When we run a container from this image, the volume will be initialized along with any data in the specified location
- If we want to setup default files in the volume folder, the folder and file must be created first

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y vim \
    wget

RUN mkdir /data/myvol -p && \
    echo "hello world" > /data/myvol/testfile
VOLUME ["/data/myvol"]
```



EX8.8 – Volumes in Dockerfile

1. Open your Dockerfile in the `myimage` folder
2. Delete or comment out the `CMD` and `ENTRYPOINT` instructions
3. Add an instruction to create a folder `/data/myvol`
`RUN mkdir /data/myvol -p`
4. Add an instruction to create a file called `test` inside `/data/myvol`
`RUN echo "put some text here" > /data/myvol/test`
5. Define a volume at `/data/myvol`
`VOLUME /data/myvol`
6. Build the image and tag as `1.1`
`docker build -t myimage:1.1 .`
7. Run a container from the image and specify a bash terminal as your process
8. Check the `/data/myvol` folder and verify that your file is present



Data containers

- A data container is a container created for the purpose of referencing one or many volumes
- Data containers don't run any application or process
- Used when you have persistent data that needs to be shared with other containers
- When creating a data container, you should give it a custom name to make it easier to reference



EX8.7 – Rename some containers

1. Run a container with a custom name of “mycontainer”
2. Rename the container to “testcontainer”



Creating data containers

- We just need to run a container and specify a volume
- Should run a container using a lightweight image such as busybox
- No need to run any particular process, just run “true”

Run our data container using the busybox image

```
docker run --name mydata -v /data/app1 busybox true
```



Using data containers

- Data containers can be used by other containers via the `--volumes-from` option in the `docker run` command
- Reference your data container by its container name. For example:
`--volumes-from` `datacontainer ...`

```
1 $ docker run --name appdata -v /data/app1 busybox
2 65d22a45d8ca11d77eed0faa8689d511a2265de1790e9bc41302378d0ac93c75
3
4 johnnytu@docker-ubuntu:~$ docker run -it --volumes-from appdata ubuntu:14.04
5 root@4ef756eeb298:#
6 root@4ef756eeb298:# cd /data/app1/
7 root@4ef756eeb298:/data/app1#
```



EX8.8 – Data containers

1. Run a busybox container and define a volume at /srv/www. Call the container “webdata”

```
docker run --name webdata -v /srv/www busybox
```

2. Run an ubuntu container and reference the volume in your webdata container. Name it webserver

```
docker run -it --name webserver --volumes-from webdata  
ubuntu:14.04
```

3. On the container terminal, verify that you can see the /srv/www folder

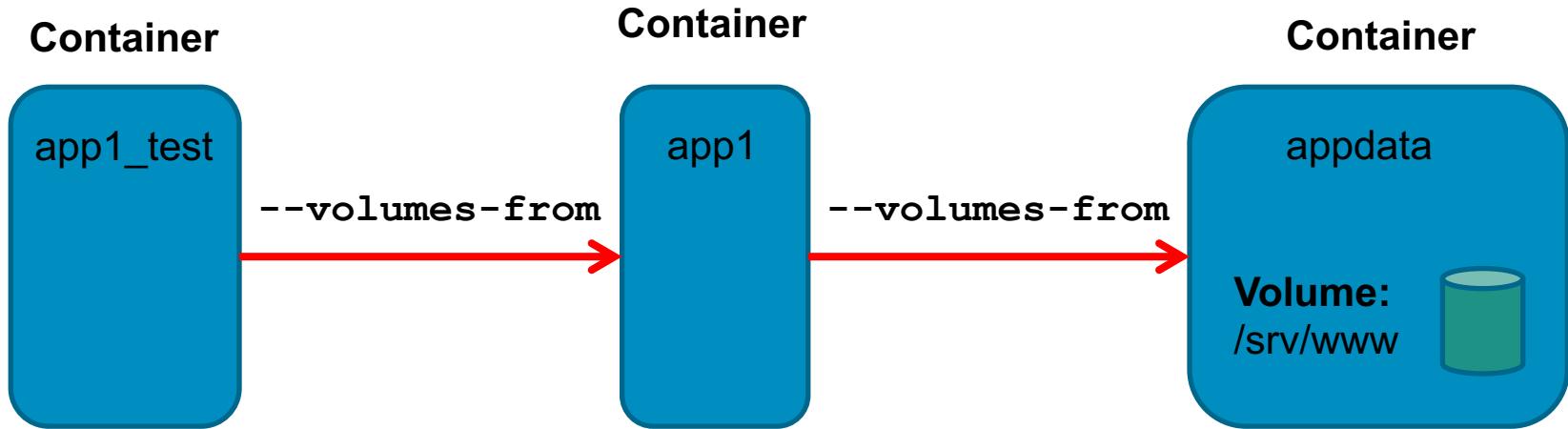
4. Add a text file to the folder

5. Exit the terminal but do not stop the container

```
CTRL + P + Q
```



Mounting volumes from other containers



- Container `app1` will mount the `/srv/www` volume from `appdata`
- Container `app1_test` will mount all volumes in `app1`, which will be the same `/srv/www` volume from `appdata`



EX8.9 – Referencing container volumes

1. Run another `ubuntu` container and mount all the volumes from the “`webserver`” container that was created in exercise 8.8. Call this **container** `webserver2`

```
docker run -it --name webserver2 --volumes-from  
webserver ubuntu:14.04
```

2. In the container terminal, check the `/srv/www` folder and verify that the file you created in exercise 8.8 is present
3. Create another text file in the folder
4. Go back into the `webserver` container and check the `/srv/www` folder there to verify the new file from question 3) is present

```
docker attach webserver
```



A more practical example

- Let's run an NGINX container but separate its operation across multiple containers
 - One container to handle the log files
 - One container to handle the web content that is to be served
 - One container to run NGINX



Setup the data containers

- Setup the logdata container

```
$ docker run --name logdata -v /var/log/nginx busybox
```

- Setup the webdata container, mount it to our public_html folder

```
$ docker run --name webdata \
-v /home/johnnytu/public_html:/usr/share/nginx/html busybox
```



Setup the webserver container

- This container needs to run in detached mode
- We can specify the `--volumes-from` option multiple times
- Remember to use `-P` for port mapping

```
$ docker run --name webserver -d -P \
  --volumes-from webdata \
  --volumes-from logdata nginx \
```



EX8.10 – Build the NGINX example

1. Double check that your `public_html` folder contains an `index.html` file
2. Create one if otherwise
3. Follow the steps on the previous slides to create your data containers first
4. Then create the `webcontainer` and mount the volumes from both data containers
5. Run `docker ps` to find the port mapping for port 80 for your webserver container. Note down this number somewhere.



EX8.10 – (cont'd)

5. Get terminal access into webcontainer and check the /usr/share/nginx/html and /var/log/nginx folders.
docker exec -it webcontainer bash
6. Check that the index.html file matches the one in your public_html folder on the host
7. Go to /var/log/nginx, open and follow the access log
tail -f /var/log/nginx/access.log
8. On your browser, go to the URL of your AWS instance and specify the port number
9. Verify that you can see the contents of your index.html page in public_html
10. Hit the page a few more times and check for log entries on your terminal



EX8.10 – (cont'd)

11. Exit the `webcontainer` terminal and return to your host terminal
12. Modify the contents of your `index.html` inside `public_html` and hit the server URL on your browser again to confirm that the changes have been picked up.



Backup your data containers

- It's a good idea to back up data containers such as our logdata container, which has our NGINX log files
- Backups can be done with the following process:
 - Create a new container and mount the volumes from the data container
 - Mount a host directory as another volume on the container
 - Run the tar process to backup the data container volume onto your host folder

```
$ docker run --volumes-from logdata \
    -v /home/johnnytu/backups:/backup \
    ubuntu:14.04 \
    tar cvf /backup/nginxlogs.tar /var/log/nginx
```



EX8.11 – backup our log container

1. Create a folder called `backup` in your home directory
2. Backup the volume in your `logdata` container with the following command (replace the “`johnnytu`” with your username and put the whole command on one line)

```
$ docker run --volumes-from logdata \
-v /home/johnnytu/backups:/backup \
ubuntu:14.04 \
tar cvf /backup/nginxlogs.tar /var/log/nginx
```

3. Check your `backup` folder for the tar file
4. Run `tar -tvf nginxlogs.tar` and verify that you can see the files `error.log` and `access.log`



Volumes defined in images

- Most images will have volumes defined in their Dockerfile
- Can check by using `docker inspect` command against the image
- `docker inspect` can be run against an image or a container
- To run against an image, specify either the image repository and tag or the image id.

Inspect the properties of the `ubuntu:14.04` image

```
docker inspect ubuntu:14.04
```

OR

```
docker inspect <image id>
```



Inspecting an image

```
[{"Cmd": [
    "nginx",
    "-g",
    "daemon off;"],
  "Image": "d8a70839d9617b3104ac0e564137fd794fab7c71900f6347e99fba7f3fe71a30",
  "Volumes": {
    "/var/cache/nginx": {}},
  "VolumeDriver": "",
  "WorkingDir": "",
  "Entrypoint": null,
  "NetworkDisabled": false,
  "MacAddress": "",
  "OnBuild": [],
  "Labels": {}},
  {"Architecture": "amd64",
  "Os": "linux",
  "Size": 0,
  "VirtualSize": 132881060,
  "GraphDriver": {
    "Name": "aufs",
    "Data": null}}]
```



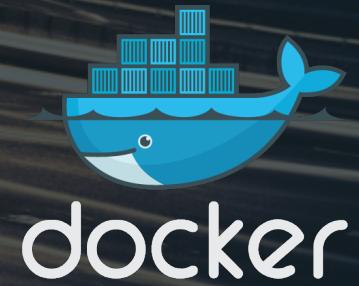
Module summary

- Volumes are created with the `docker volume create` command
- Volumes can be mounted when we run a container during the `docker run` command or in a Dockerfile
- Volumes bypass the copy on write system
- We can map a host directory to a volume in a container
- A volume persists even after its container has been deleted



Module 9:

Container Networking



Module objectives

In this module we will

- Explain the Docker networking model for containers
- Learn how to map container ports to host ports manually and automatically
- Learn how to link containers together by using the bridge network



Docker networking model

- Containers do not have a public IPv4 address
- They are allocated a private address
- Services running on a container must be exposed port by port
- Container ports have to be mapped to the host port to avoid conflicts



The docker0 bridge

- When Docker starts, it creates a virtual interface called `docker0` on the host machine
- `docker0` is assigned a random IP address and subnet from the private range defined by RFC 1918

```
johnnytu@docker-ubuntu:~$ ip a
...
...
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/16 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
        valid_lft forever preferred_lft forever
```



The docker0 bridge

- The `docker0` interface is a virtual Ethernet bridge interface
- It passes or switches packets between two connected devices just like a physical bridge or switch
 - Host to container
 - Container to container
- Each new container gets one interface that is automatically attached to the `docker0` bridge



Checking the bridge interface

- We can use the `brctl` (bridge control) command to check the interfaces on our `docker0` bridge
- **Install** `bridge-utils` package to get the command
`apt-get install bridge-utils`
- **Run**
`brctl show docker0`

```
johnnytu@docker-ubuntu:~$ brctl show docker0
bridge name      bridge id          STP enabled     interfaces
docker0          8000.56847afe9799    no
```



Checking the bridge interface

- Spin up some containers and then check the bridge again

```
$ docker run -d -it ubuntu:14.04  
d0e7cf41df3916dc3488e32e9a3f984be1236402e3ab1940fb43af43325f605  
$ docker run -d -it ubuntu:14.04  
b6b76d93e177de7e05673556c9b9a13e86d4132dda2ee7147a417ebc18a946c4  
  
$ brctl show docker0  
bridge name      bridge id          STP enabled    interfaces  
docker0          8000.56847afe9799    no            vethdc14acd  
                           vethdd0c513
```



Check container interface

- Get into the container terminal and run the `ip a` command

```
johnnytu@docker-ubuntu:~$ docker exec -it d0e7cf41df3916dcd3 bash
root@d0e7cf41df39:#
root@d0e7cf41df39:#
root@d0e7cf41df39:## ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever
414: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:78 brd ff:ff:ff:ff:ff:ff
        inet 172.17.0.120/16 scope global eth0
            valid_lft forever preferred_lft forever
        inet6 fe80::42:acff:fe11:78/64 scope link
            valid_lft forever preferred_lft forever
root@d0e7cf41df39:#
root@d0e7cf41df39:#
```



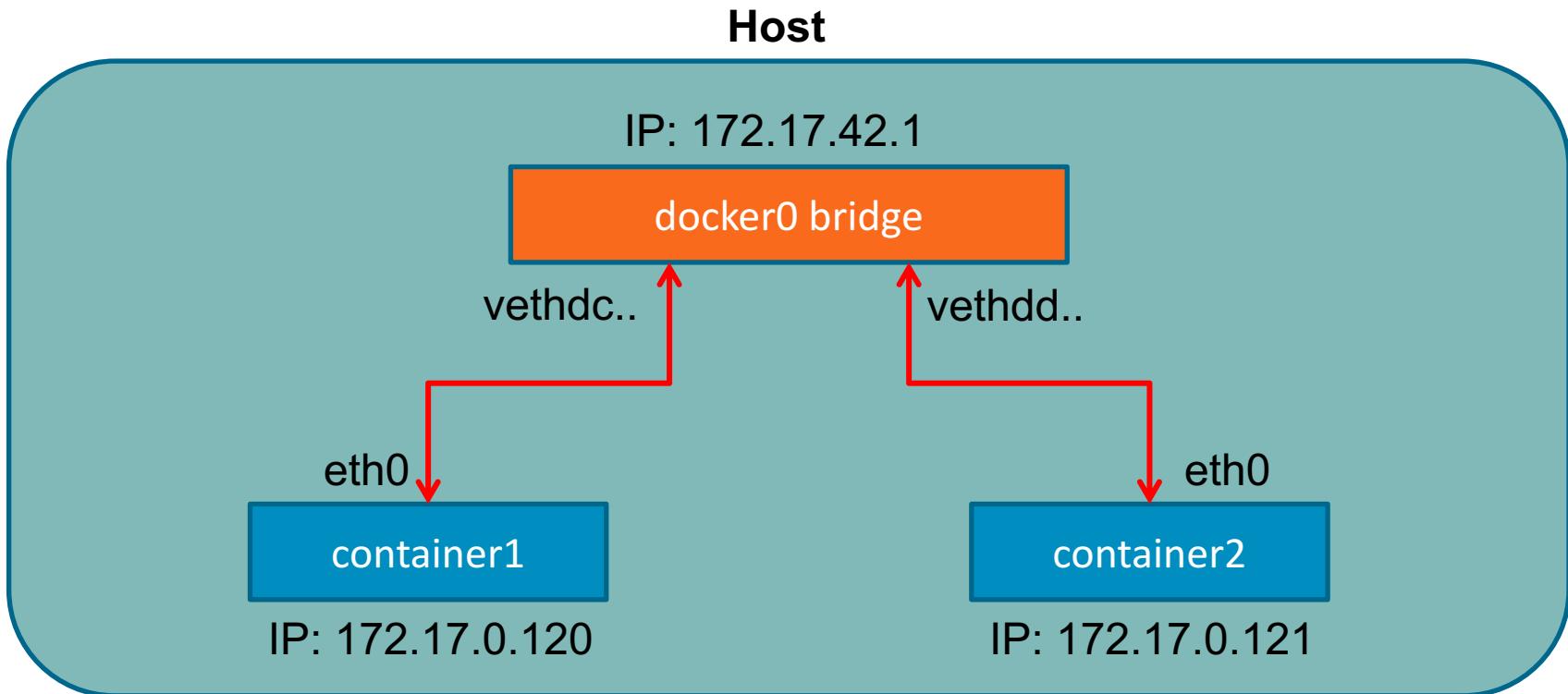
Check container networking properties

- Use `docker inspect` command and look for the `NetworkSettings` field

```
"NetworkSettings": {  
    "Bridge": "docker0",  
    "Gateway": "172.17.42.1",  
    "GlobalIPv6Address": "",  
    "GlobalIPv6PrefixLen": 0,  
    "IPAddress": "172.17.0.120",  
    "IPPrefixLen": 16,  
    "IPv6Gateway": "",  
    "LinkLocalIPv6Address": "fe80::42:acff:fe11:78",  
    "LinkLocalIPv6PrefixLen": 64,  
    "MacAddress": "02:42:ac:11:00:78",  
    "PortMapping": null,  
    "Ports": {}  
},
```



Diagram of networking model



Docker network command

- The `docker network` command and subcommands allow us to interact with Docker Networks and the containers in them
- Subcommands are:
 - `docker network create`
 - `docker network connect`
 - `docker network ls`
 - `docker network rm`
 - `docker network disconnect`
 - `docker network inspect`



Default networks

- Docker comes with three networks automatically setup
- View the networks by using the docker network ls command
- The bridge network is the docker0 bridge
- By default all containers are connected to the bridge network

```
student@dockerhost:~$ docker network ls
NETWORK ID      NAME      DRIVER
d1dc8ce401a4    bridge    bridge
123297c4b101    none     null
ec95beed1ab7    host     host
```



Connecting to another network

- Containers can be launched on another network with the `--net` option in the `docker run` command
- Connecting the container to the `none` network will add it into its own network stack. The container will not have a network interface
- Connecting to the `host` network adds the container to the same network stack as the host

Launch an NGINX container on the host network

```
docker run -d --net=host nginx
```



Inspecting a network

- Use the docker network inspect command to display the details of a particular network
- Will also show the containers connected to that network

```
student@dockerhost:~$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "d1dc8ce401a4c1f84d4276f4418f024257448c6463a88eb6d1c75a00ba5bc525",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
            ]
        },
        "Containers": {
            "7886c7cbd03bec775b4ceb05a745c071211fb790d556380800d3337acde85f23": {
                "EndpointID": "db153400ad297726bf9fc34ffb978b66ad059a54217d65963965676a4240268",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        }
    }
]
```



EX9.1 – Using the default bridge

1. Make sure there are no existing containers running
`docker stop $(docker ps -aq)`
2. Launch an Ubuntu container called `ubuntu1` in detached mode
`docker run --name=ubuntu1 -d -it ubuntu:14.04`
3. Inspect the bridge network and note down the container IP address
`docker network inspect bridge`
4. Launch another Ubuntu called `ubuntu2` container and get terminal access inside
`docker run --name=ubuntu2 -it ubuntu:14.04 bash`
5. Ping your `ubuntu1` using the IP Address obtained in question 2.
6. Try and ping `ubuntu1` using the container name. You will notice how it does not work
7. Exit the container terminal without stopping it
`CTRL + P + Q`



Creating your own network

- We can setup our own network for running our containers
- Use `docker network create` command
- Two types of networks we can create
 - Bridge
 - Overlay
- A bridge network is very similar to the `docker0` network (the default bridge network we've seen)
- An Overlay network can span across multiple hosts



Creating a bridge network

- Use the `--driver` option in `docker network create` and specify the bridge driver
- Example:

```
docker network create --driver bridge my_bridge
```

```
student@dockerhost:~$ docker network create --driver bridge my_bridge
391162b1413740bf2714ca052b5347ddfa9361661c691d0e3fc991f103271a6d
student@dockerhost:~$ docker network ls
NETWORK ID      NAME      DRIVER
ec95beed1ab7    host      host
391162b14137    my_bridge  bridge
d1dc8ce401a4    bridge    bridge
123297c4b101    none     null
```



Checking container network

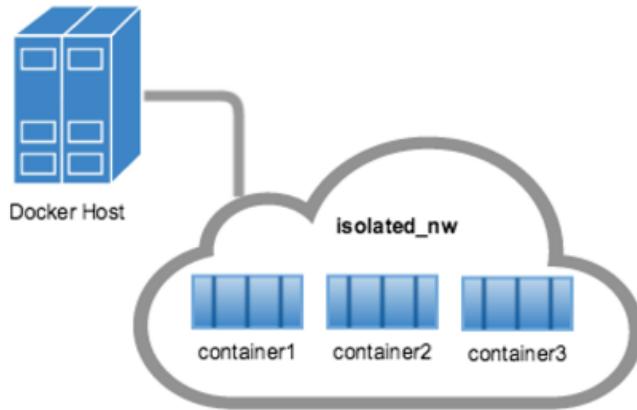
- Inspect the container and look for the Networks field in the docker inspect output
- Should indicate the Network name, Gateway, IP Address etc...

```
student@dockerhost:~$ docker run -d --net=my_bridge nginx
a0a2f1fdb867d265dfcbfb60f3f9662382711a5473322156788f5dc8b570f70
.....
student@dockerhost:~$ docker inspect awesome_wescoff
[
{
    "Id": "a0a2f1fdb867d265dfcbfb60f3f9662382711a5473322156788f5dc8b570f70",
    .....
    "Networks": {
        "my_bridge": {
            "EndpointID": "786f3633b7ae4ff4e37f63df7fc8cd4c84a5607bab6243bf8f4d80a1468c3bf",
            "Gateway": "172.18.0.1",
            "IPAddress": "172.18.0.2",
            "IPPrefixLen": 16,
            .....
        }
    }
}
```



Bridge network

- Containers must reside on the same host
- Containers in user defined bridge network can communicate with each other using their IP address and container name
 - Container must be launched with `--name` option



EX9.2 – Create a bridge network

1. Create a new bridge network called my_bridge
`docker network create --driver bridge my_bridge`
2. Verify the network was successfully created by running
`docker network ls`
3. Launch an Ubuntu container in the background on your my_bridge network. Call the container ubuntu3
`docker run -d -it --net=my_bridge --name ubuntu3 ubuntu:14.04`
4. Launch another Ubuntu container on your my_bridge network. Call the container ubuntu4 and run a bash terminal
`docker run -it --net=my_bridge --name ubuntu4 ubuntu:14.04`
5. In your ubuntu4 terminal, open the /etc/hosts file



EX9.2 – (cont'd)

6. Check for the following entry

172.18.0.2 ubuntu3

172.18.0.2 ubuntu3.my_bridge

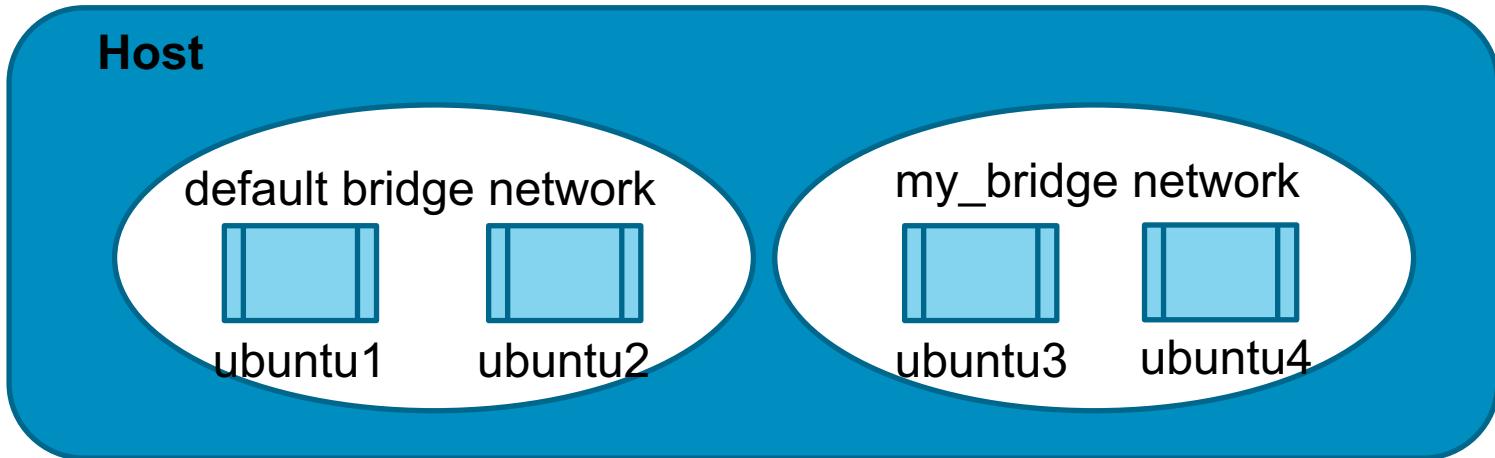
7. Ping ubuntu3 using the container name.

8. Try to ping the ubuntu1 or ubuntu2 containers from Exercise 9.1. Remember that those containers are on a different network and you will get no response



Connecting containers to multiple networks

- Containers can be connected to multiple networks via the docker network connect command
- Command syntax
docker network connect [network name] [container name]
- **Our Example at the moment:**



Connecting containers to multiple networks

- Let's connect our ubuntu2 container to the user defined my_bridge network
- Run

```
docker network connect my_bridge ubuntu2
```
- The connect command will not disconnect the container from its current network
- Inspect the container and you should see two networks listed in the output



Container inspection

```
"Networks": {  
    "bridge": {  
        "EndpointID": "4bbf9e7db81b40f98224227c02ef18f9bbbd1188aac036440d979a5c7fc2baaa",  
        "Gateway": "172.17.0.1",  
        "IPAddress": "172.17.0.3",  
        "IPPrefixLen": 16,  
        "IPv6Gateway": "",  
        "GlobalIPv6Address": "",  
        "GlobalIPv6PrefixLen": 0,  
        "MacAddress": "02:42:ac:11:00:03"  
    },  
    "my_bridge": {  
        "EndpointID": "d1a77a794c12699a6fe6b5e06213dfab4990c81b6a1943c2c9f82768df7d0850",  
        "Gateway": "172.18.0.1",  
        "IPAddress": "172.18.0.4",  
        "IPPrefixLen": 16,  
        "IPv6Gateway": "",  
        "GlobalIPv6Address": "",  
        "GlobalIPv6PrefixLen": 0,  
        "MacAddress": "02:42:ac:12:00:04"  
    }  
}
```



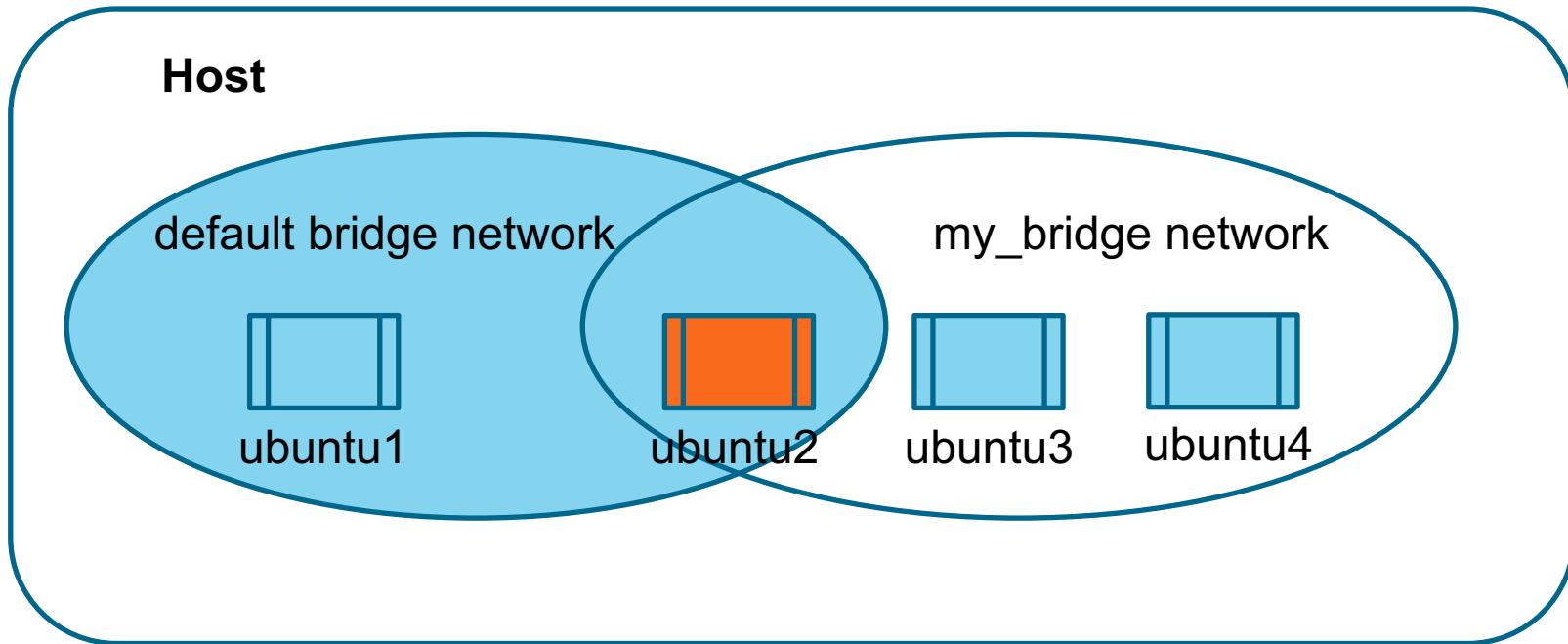
Checking the /etc/hosts file

- In the ubuntu2 container, the /etc/hosts file should now have an entry for ubuntu3 and ubuntu4
- You should also see an entry for ubuntu2 in the /etc/hosts file on ubuntu3 and ubuntu4

```
root@2ed145df96f5:/# cat /etc/hosts
172.17.0.3      2ed145df96f5
127.0.0.1      localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.18.0.2      ubuntu3
172.18.0.2      ubuntu3.my_bridge
172.18.0.3      ubuntu4
172.18.0.3      ubuntu4.my_bridge
```



Our host network



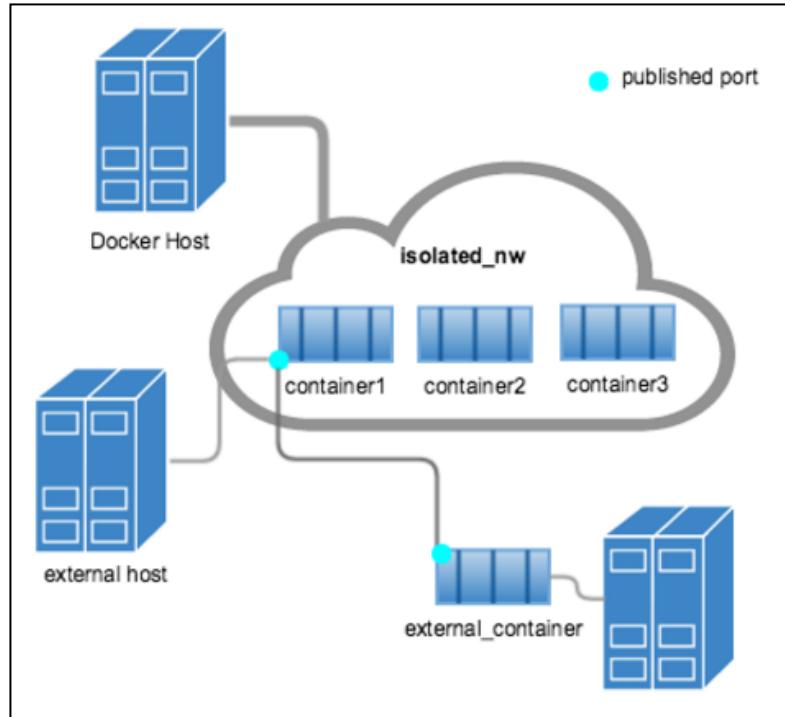
EX9.3 – Connect to multiple networks

1. Connect your ubuntu2 container to the my_bridge network
docker network connect my_bridge ubuntu2
2. Check that you can ping the ubuntu3 and ubuntu4 containers
docker exec ubuntu2 ping ubuntu3
3. Check that you can ping the ubuntu2 and ubuntu4 containers
docker exec ubuntu3 ping ubuntu2



Exposing a container in a bridge network

- Containers running in a bridge network can only be accessed by the host in which that network resides
- To make a container accessible to the outside we must expose the container ports and map them to a port on the host.
- The container can be accessed via the mapped host port



Mapping ports

- Map exposed container ports to ports on the host machine
- Ports can be manually mapped or auto mapped
- You can see the port mapping for each container on the docker ps output

CONTAINER ID	IMAGE	COMMAND	PORTS	NAMES
edb8d8cea278	mynginx:latest	"nginx -g 'daemon off'; ..."	0.0.0.0:80->80/tcp, 0.0.0.0:8080->8080/tcp, 443/tcp	mad_newton
661389e77b83	nginx:latest	"nginx -g 'daemon off'; ..."	80/tcp, 443/tcp	boring_ardinghelli
b6b76d93e177	ubuntu:14.04	"/bin/bash"	...	gloomy_wright
d0e7cf41df39	ubuntu:14.04	"/bin/bash"	...	kickass_pare



Manual port mapping

- Uses the -p option (smaller case p) in the docker run command
- Syntax
 - p [host port]:[container port]
- To map multiple ports, specify the -p option multiple times

Map port 8080 on the tomcat container to port 80 on the host

```
docker run -d -p 80:8080 tomcat
```

Map port 80 on the host to port 80 on the nginx container and port 81 on the host to port 8080 on the nginx container

```
docker run -d -p 80:80 -p 81:8080 nginx
```



Docker port command

- docker ps output is not very ideal for displaying port mappings
- We can use the docker port command instead

```
johnnytu@docker-ubuntu:~$ docker port mad_newton  
80/tcp -> 0.0.0.0:80  
8080/tcp -> 0.0.0.0:8080
```



EX9.4 – Manual port mapping

1. Run an nginx container and map port 80 on the container to port 80 on your host. Map port 8080 on the container to port 90 on the host
`docker run -d -p 80:80 -p 90:8080 nginx`
2. Verify the port mappings with the docker port command
`docker port <container name>`



Automapping ports

- Use the `-P` option in `docker run` command
- Automatically maps exposed ports in the container to a port number in the host
- Host port numbers used go from 49153 to 65535
- Only works for ports defined in the Dockerfile `EXPOSE` instruction

Auto map ports exposed by the NGINX container to a port value on the host

```
docker run -d -P nginx:1.7
```



EXPOSE instruction

- Configures which ports a container will listen on at runtime
- Ports still need to be mapped when container is executed

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y nginx

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```



EX9.5 – Auto mapping

1. Run a container using the “myimage” image built from previous exercises. Use the `-P` option for auto mapping ports
`docker run -d -P myimage`
2. Check the port mappings. Notice that there are none
3. Modify the Dockerfile inside myimage and expose port 80 and 8080.
`EXPOSE 80 8080`
4. Build the image
`docker build -t myimage .`
5. Repeat step 1 and 2. This time you should notice your port mapping



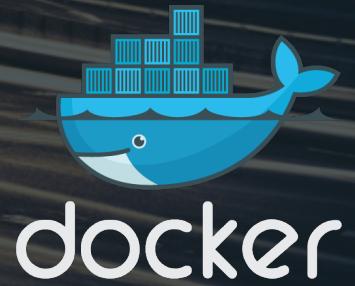
Module summary

- Docker containers run in a subnet provisioned by the docker0 bridge on the host machine
- We can create our own bridge or overlay network to run containers on
- Auto mapping of container ports to host ports only applies to the port numbers defined in the Dockerfile EXPOSE instruction
- Key Dockerfile instructions
 - EXPOSE



Module 10:

Docker in Continuous Integration



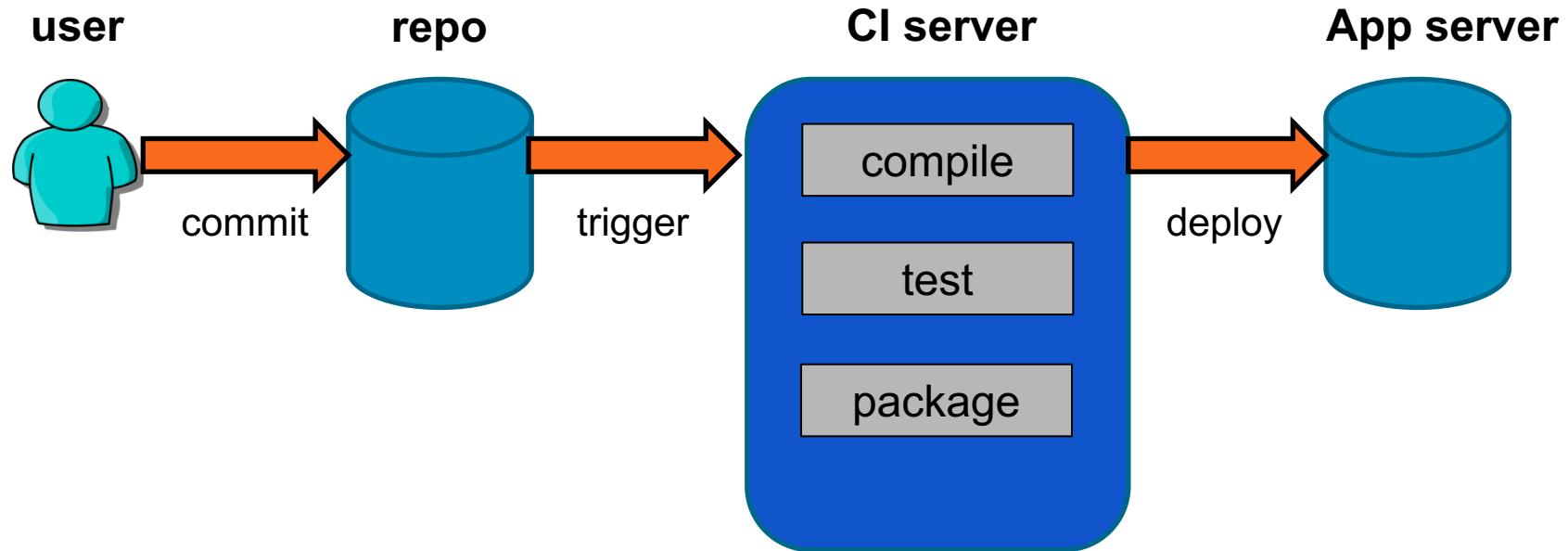
Module objectives

In this module we will:

- Explore ways we can fit Docker containers into our continuous integration process
- Setup an automated build in Docker Hub

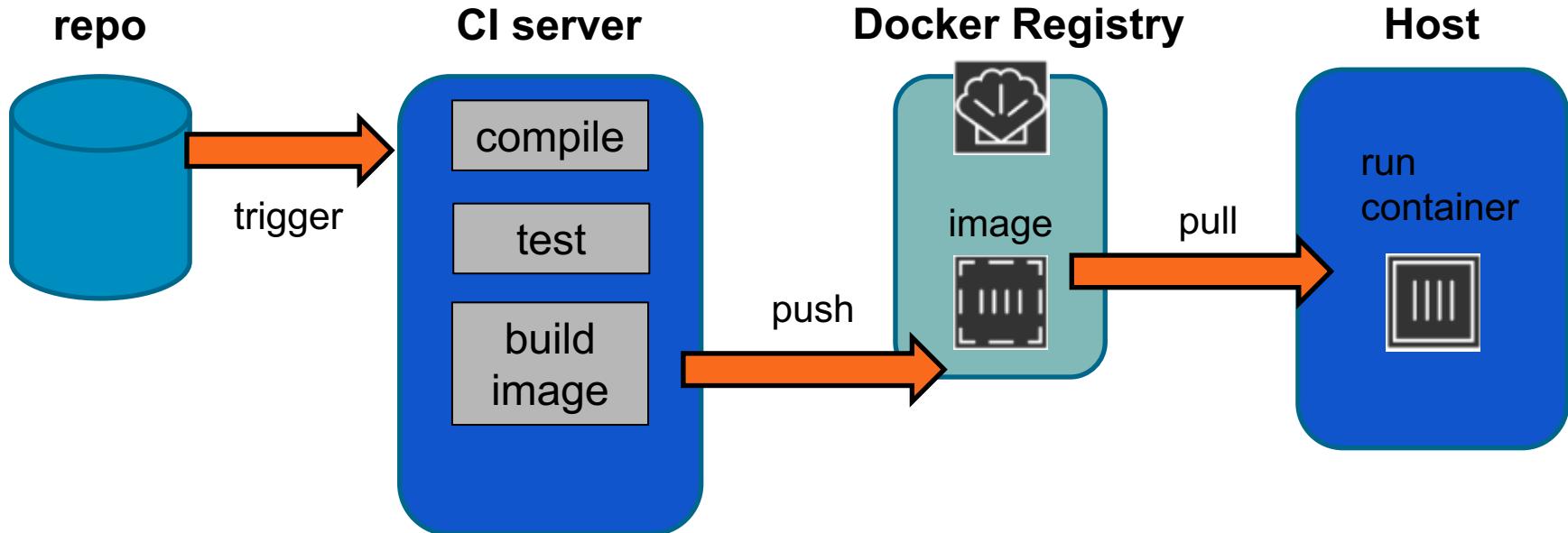


Traditional Continuous Integration



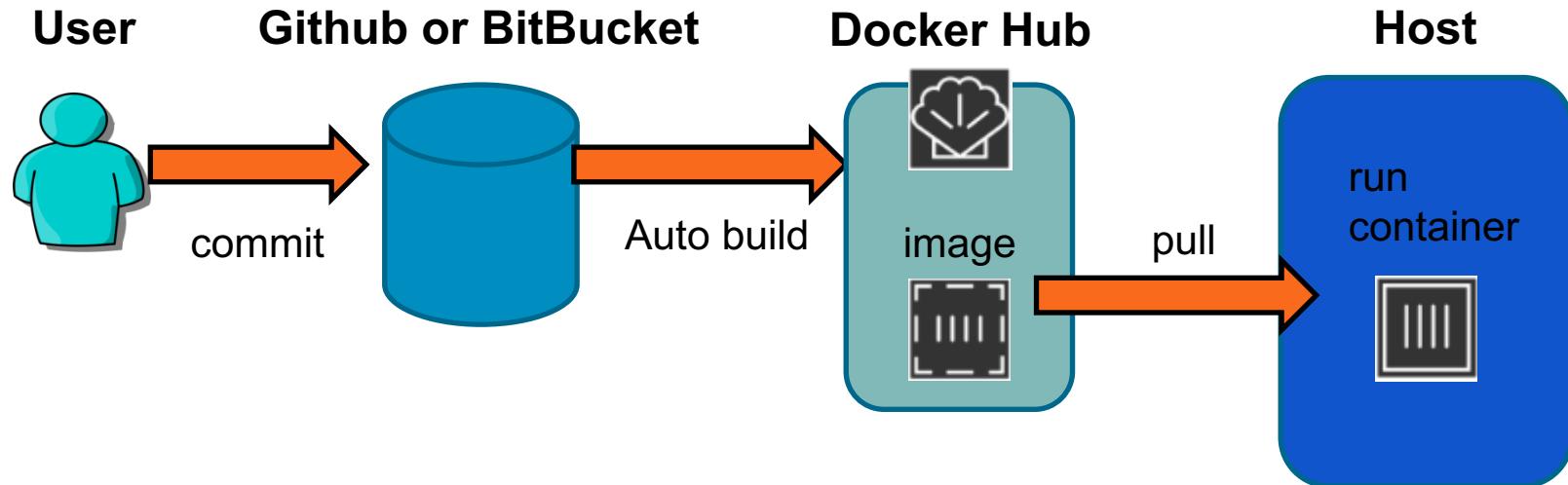
Using Docker in CI

- CI server builds Docker image and pushes into Docker Hub



Docker Hub Auto Build

- Docker Hub detects commits to source repository and builds the image
- Container is run during image build
- Testing done inside container



Setup an auto build example

- **Revision:** remember the simple “hello world” java application we built earlier?
- Let’s take that application and put it into a simple CI process using the Docker Hub auto build feature
- We will need to do the following:
 - Put our code into a repository (GitHub)
 - Setup an automated build on Docker Hub and have it connected to our GitHub account



EX10.1 – Setup GitHub account

1. Go to <https://github.com/join> and setup a GitHub account. If you have an existing GitHub account, you can choose to use that one instead.

Join GitHub

The best way to design, build, and ship software.

Step 1: Set up a personal account Step 2: Choose your plan Step 3: Go to your dashboard

Create your personal account

Username

This will be your username — you can enter your organization's username next.

Email Address

You will occasionally receive account related emails. We promise not to share your email with anyone.

Password

Use at least one lowercase letter, one numeral, and seven characters.

You'll love GitHub

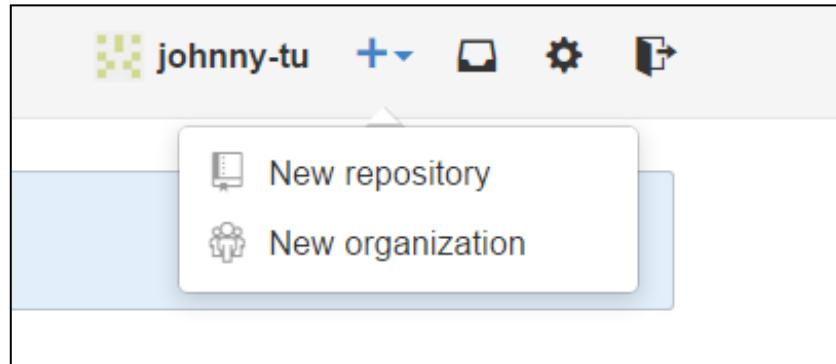
Unlimited collaborators
Unlimited public repositories

- ✓ Great communication
- ✓ Friction-less development
- ✓ Open source community



Create a new GitHub repository

- We will need a new repo for our application
- Repository name will be `javadelloworld`



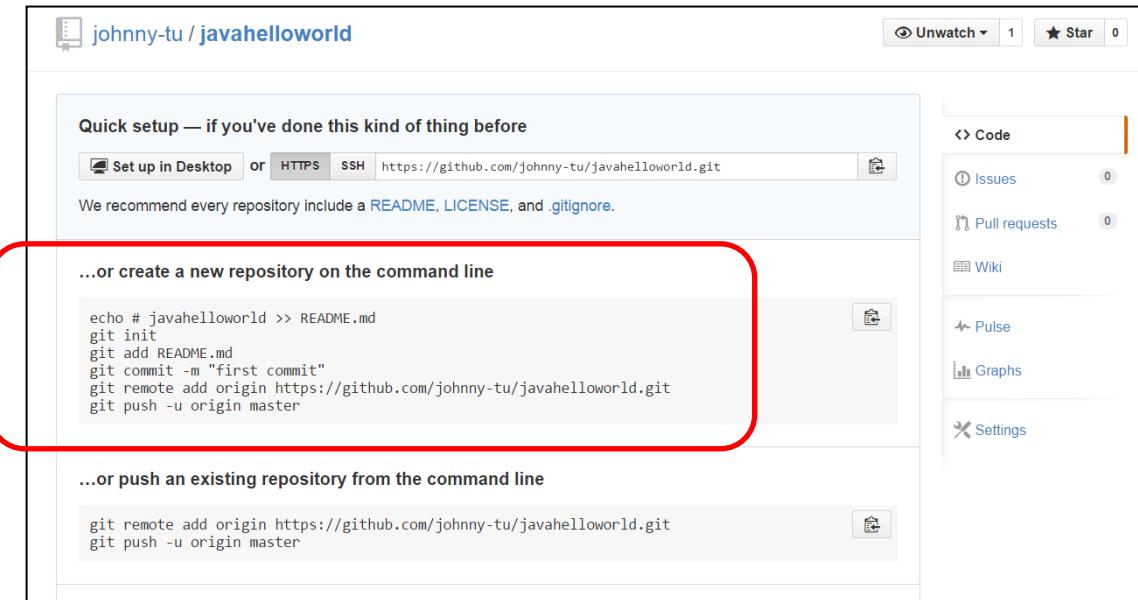
EX10.2 – Create new repository

1. In your GitHub account, create a new repository called `javadelloworld`. Make the repository public.



Add existing code to GitHub repository

- Once you have created the repository, you will see a section on the page that contains step by step instruction on adding your existing code to the repository



Push our code into the repository

- First initialise the git repository in our working directory
`git init`
- Then add the files we want to commit to the git staging area
`git add src/HelloWorld.java`
- Commit the code locally
`git commit -m "first commit"`
- Add our GitHub repository as a remote repository
`git remote add origin
https://github.com/<username>/<repo name>.git`
- Push our commit to the remote repository
`git push origin master`



EX10.3 – Push our code

1. Change directory into the `javadelloworld` folder

2. initialise the git repo

```
git init
```

3. Add the `HelloWorld.java` src file and the Dockerfile to the staging area

```
git add src/HelloWorld.java Dockerfile
```

4. Commit the code locally

```
git commit -m "first commit"
```

5. Add your GitHub repository as a remote repository

```
git remote add origin
```

```
https://github.com/<username>/<repo name>.git
```

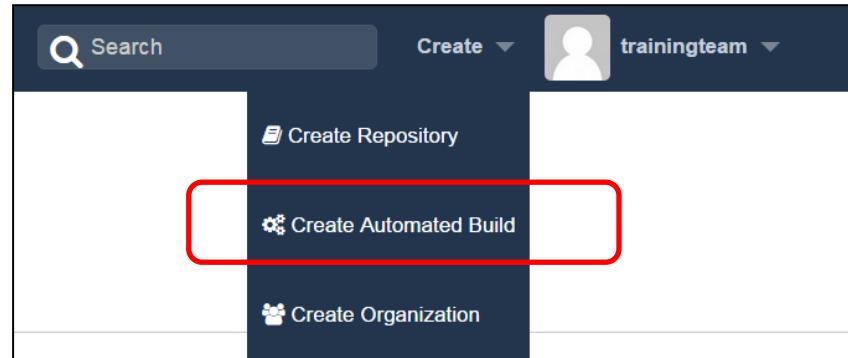
6. Push the commit to the remote repository

```
git push origin master
```



Setup Docker Hub auto build

- Click “Create Automated Build”
- Link your Docker Hub account to GitHub or Bitbucket, if you have not configured this in your settings



You haven't linked to GitHub or Bitbucket yet.

[Link Accounts](#)



Select the repository provider

- Select GitHub or BitBucket and follow the screen prompts



Choose your GitHub repository

- Once linked, you will need to click “Create Automated Build” again
- If you have multiple GitHub accounts connected to DockerHub, they will appear on this screen
- Choose the right GitHub account and then choose the “javahelloworld” repository

The screenshot shows the DockerHub interface for linking GitHub accounts. At the top, there are two buttons: "GitHub (johnny-tu)" and "Link Accounts". Below this, a section titled "Users/Organizations" lists "docker-training" and "johnny-tu". To the right, a search bar says "Type to filter" with results "HelloRedis", "inventory-service", and "javahelloworld".

Users/Organizations	Type to filter
docker-training	HelloRedis
johnny-tu >	inventory-service
	javahelloworld



Create the automated build

- By default the repository name of the automated build will be the same as the source code repository name.
- You can choose to run automated builds based on branches or tags
- The “Docker Tag Name” field specifies what tag newly built images are given

The screenshot shows a configuration interface for a Docker build. At the top, there is a dropdown menu set to "trainingteam" and a text input field containing "javahelloworld". Below this is a "Short Description (100 Characters)" text area. The main configuration section has four columns: "Type", "Name", "Dockerfile Location", and "Tag". Under "Type", a dropdown menu is set to "Branch" with a "master" option selected. The "Name" column contains "javahelloworld". The "Dockerfile Location" column has a "/" placeholder. The "Tag" column contains "latest". To the right of the "Tag" column is a blue "+" button. The entire interface is contained within a light gray box with a thin black border.



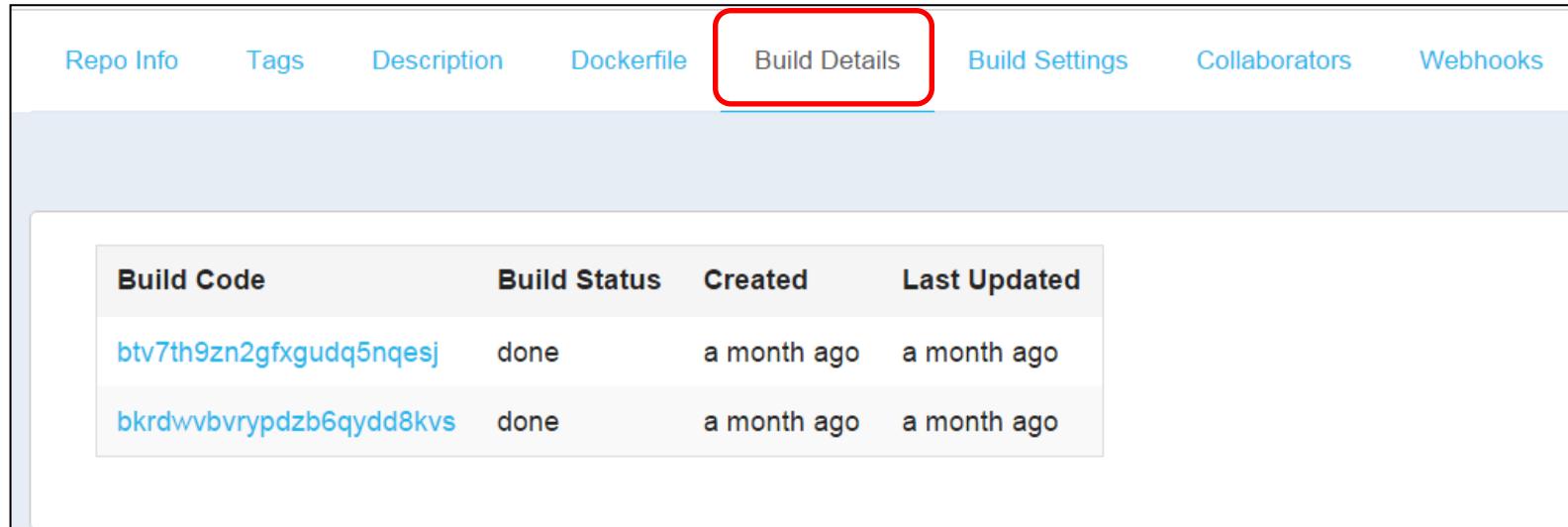
EX10.4 – Create automated build

1. Go to your Docker Hub account and setup an automated build following the steps shown in the previous slides
2. Name your auto build `javadelloworld_prod`
3. Leave the “Dockerfile Location” and Docker Tag name on their default fields. Make sure the “Active” checkbox is ticked



Checking progress and results

- An automated build repository in Docker Hub contains a “Build Details” tab
- The “Build Details” tab shows the history of the image being built



The screenshot shows a screenshot of a Docker Hub repository page. At the top, there is a navigation bar with tabs: Repo Info, Tags, Description, Dockerfile, Build Details, Build Settings, Collaborators, and Webhooks. The 'Build Details' tab is highlighted with a red box. Below the navigation bar, there is a large, empty light-gray rectangular area. Underneath this area, there is a table with a white background and a thin gray border. The table has four columns with headers: Build Code, Build Status, Created, and Last Updated. There are two rows of data in the table. The first row contains the build code [btv7th9zn2gfgudq5nqesj](#), status [done](#), created [a month ago](#), and last updated [a month ago](#). The second row contains the build code [bkrdwvbvrypdzb6qydd8kvs](#), status [done](#), created [a month ago](#), and last updated [a month ago](#).

Build Code	Build Status	Created	Last Updated
btv7th9zn2gfgudq5nqesj	done	a month ago	a month ago
bkrdwvbvrypdzb6qydd8kvs	done	a month ago	a month ago



Build log

- Click on the build id on the Build History tab to view the details of that build, including the build log

PUBLIC | AUTOMATED BUILD

trainingteam/javahelloworld-prod ☆

Last pushed: a month ago

Repo Info Tags Description Dockerfile Build Details Build Settings Collaborators Webhooks [Delete Repository](#)

Name	Value
error	
readme_contents	
created_at	a month ago
build_path	/
docker_tag	latest
source_url	https://github.com/johnny-tu/javahelloworld.git
build_code	b7v7th9zn2gfgudq5nqesj
source_branch	master

[Trigger a Build](#) [Source Project](#)

DOCKER PULL COMMAND

```
docker pull trainingteam/javahelloworld-prod
```

DESCRIPTION

This is our auto build

OWNER



EX10.5 – Push changes and build again

1. Check the Build Status tab of the automated build you created in Exercise 10.4
2. Notice that the build has already started
3. Edit some of the source code in `HelloWorld.java` (You don't need actual code, just change the line which prints "hello world" to print something else)
4. Build the image locally and verify your code change
5. Commit the code and push to GitHub
6. Wait a little and then check the Docker Hub automated build page again
7. Notice the additional build id's

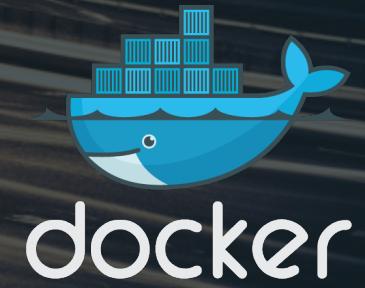


Module summary

- There are many ways for Docker containers to fit into your continuous integration or continuous delivery process
- Docker Hub's auto build repository is one way for us to build and distribute production ready images



Further Information



Additional resources

- Docker homepage - <http://www.docker.com/>
- Docker Hub - <https://hub.docker.com>
- Docker blog - <http://blog.docker.com/>
- Docker documentation - <http://docs.docker.com/>
- Docker code on GitHub - <https://github.com/docker/docker>
- Docker mailing list - <https://groups.google.com/forum/#!forum/docker-user>
- Docker on IRC: irc.freenode.net and channels #docker and #docker-dev
- Docker on Twitter - <http://twitter.com/docker>
- Get Docker help on Stack Overflow -
<http://stackoverflow.com/search?q=docker>



Where to next?

- Take the Advanced Docker Topics course
- Agenda
 - Controlling the Docker daemon
 - Security and TLS
 - Multi-host Networking
 - Docker Content Trust
 - Setting up your own Registry
 - Docker Trusted Registry
 - Docker Machine
 - Docker Swarm
 - Building micro service applications
 - Docker Compose





THANK YOU