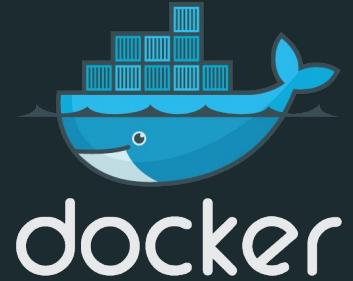


Docker Fundamentals

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



Instructor Intro



Agenda

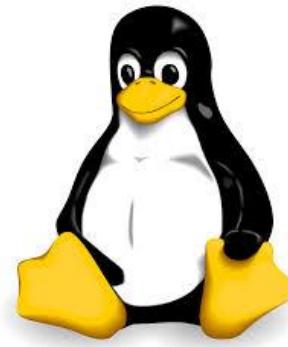
- 9-9:15 Introduction and Environment Setup
- 9:15-9:30 Images and Containers
- 9:30-10 Building Images
- 10-10:30 Docker and CICD
- 10:30-10:45 Break
- 10:45-11:00 What's new in Docker?
- 11:00-11:15 Container Volumes
- 11:15-11:45 Container Networking
- 11:45-12 Docker Content Trust / Notary
- 12-12:45 Lunch
- 12:45 - 1:15 Docker Trusted Registry 1.4
- 1:15 - 1:45 Docker Swarm 1.0
- 2 - 4 Tutum & UCP Overview
- 4- 4:30 Docker Compose 1.5
- 4:30- 5:30 Docker Orchestration Lab (#2 on github.com/dceu_tutorials)
- 5:30 - 6 Open Forum / Labs Followups / Q+A



Pre-requisites

Prerequisites

- Basic Docker Experience
- SSH (or PuTTY)
- Be familiar with Linux command line
- Ask questions at anytime



Your training environment

- You should have been provided with seven Ubuntu 14.04 instances
 - node-0
 - node-1
 - node-2
 - node-3
 - ducp-0
 - ducp-1
 - ducp-2
- **Your instructor will provide the login details**
- Some optional exercises can be done on your own PC or Mac



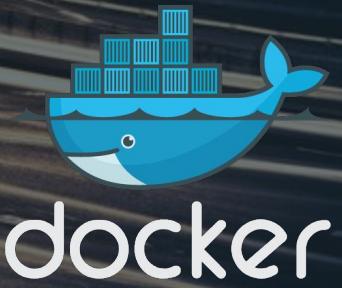
Access your training environment

Login to your 4 Amazon AWS instances using the credentials provided to you by the instructor.

- For MAC or Linux users, use SSH on your terminals
- For PC users, use Putty



Introduction to Containers



Module objectives

In this module we will:

- 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

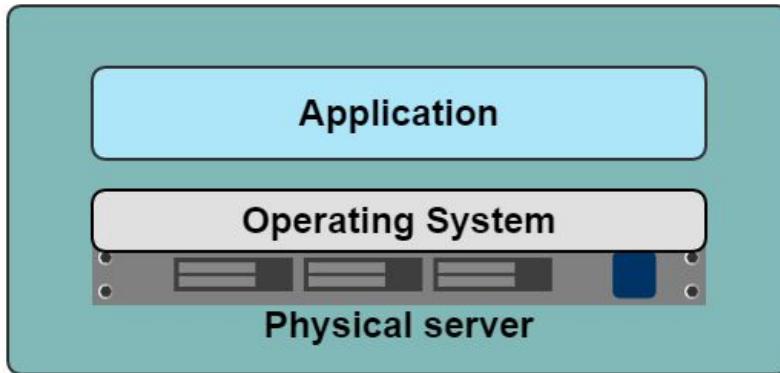
- The Docker Platform consists of multiple products/tools
 - Docker Engine
 - Docker Hub
 - Docker Trusted Registry
 - Docker Machine
 - Docker Swarm
 - Docker Compose
 - Kitematic



A History Lesson

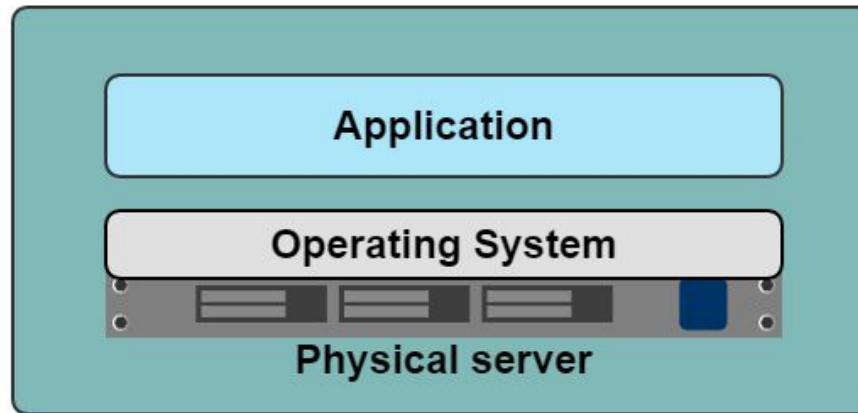
In the Dark Ages

One application on one physical server



Historical limitations of application deployment

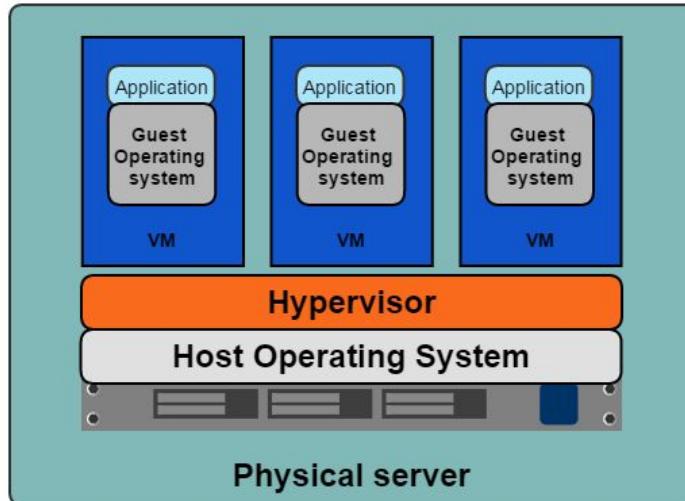
- Slow deployment times
- Huge costs
- Wasted resources
- Difficult to scale
- Difficult to migrate
- Vendor lock in



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



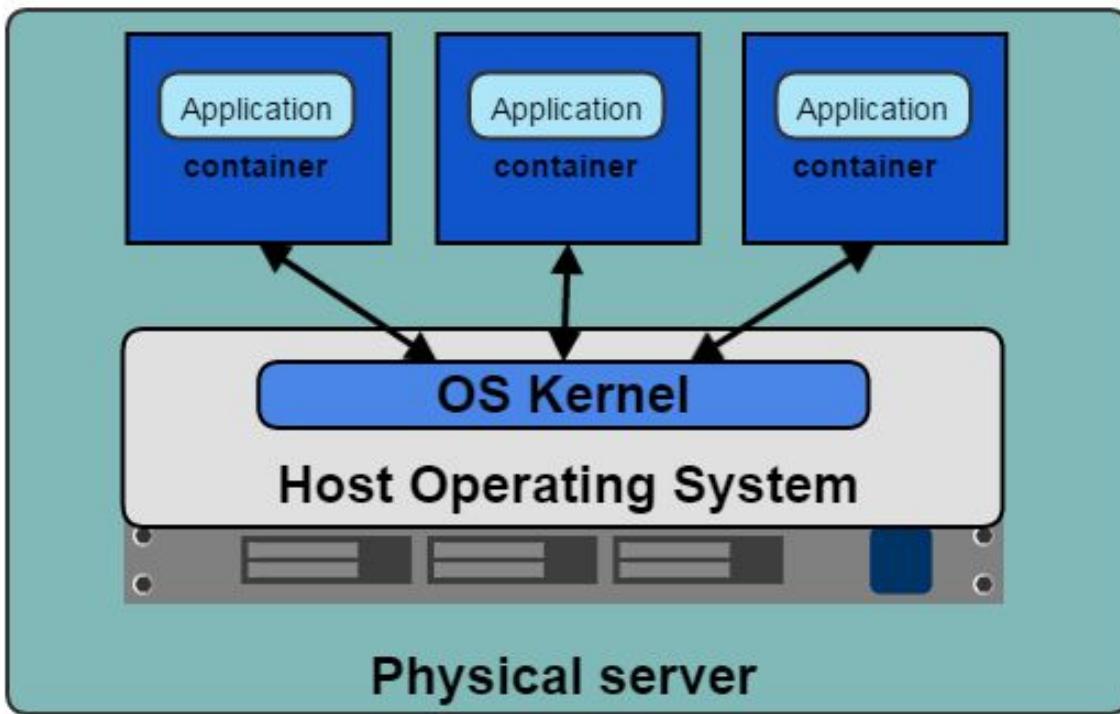
Introducing Containers

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

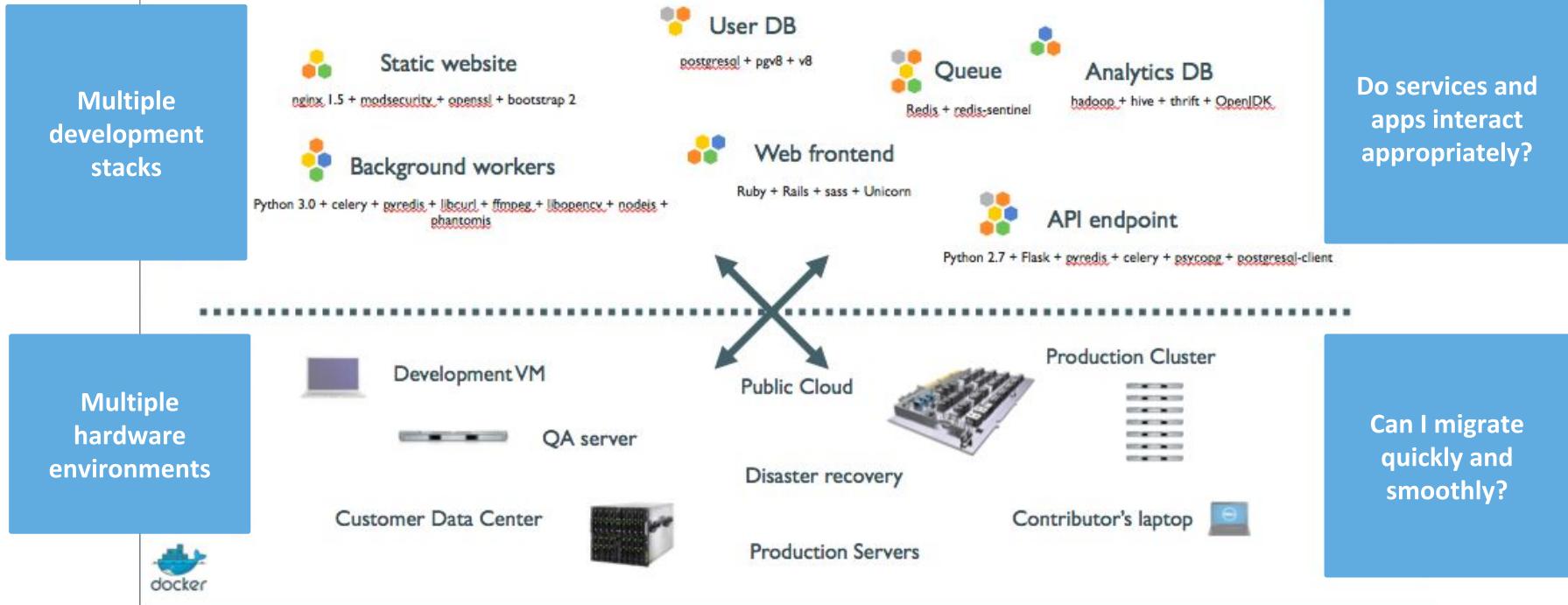


Why use Docker?

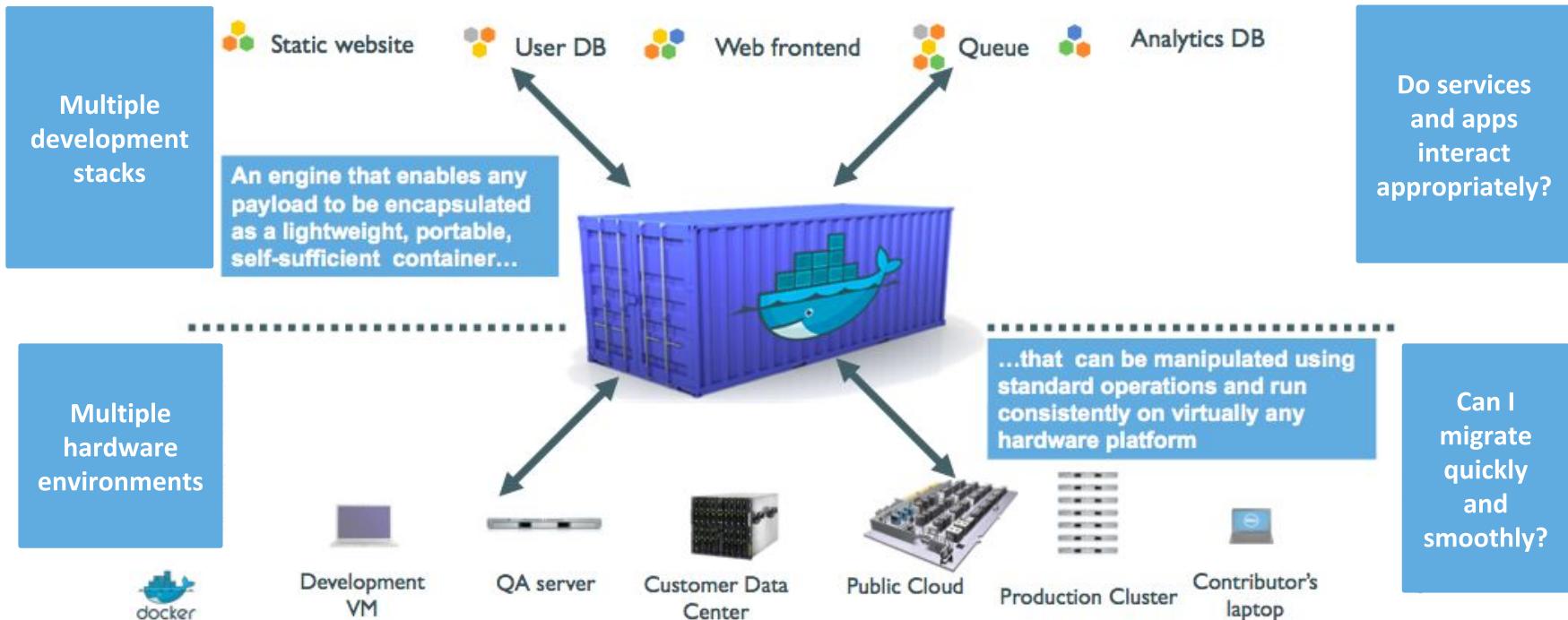
- Applications are no longer one big monolithic stack
- Service oriented architecture means there are multiple application stacks that need to be deployed
- Services are decoupled, built iteratively and scaled out
- Deployment can be a complex exercise



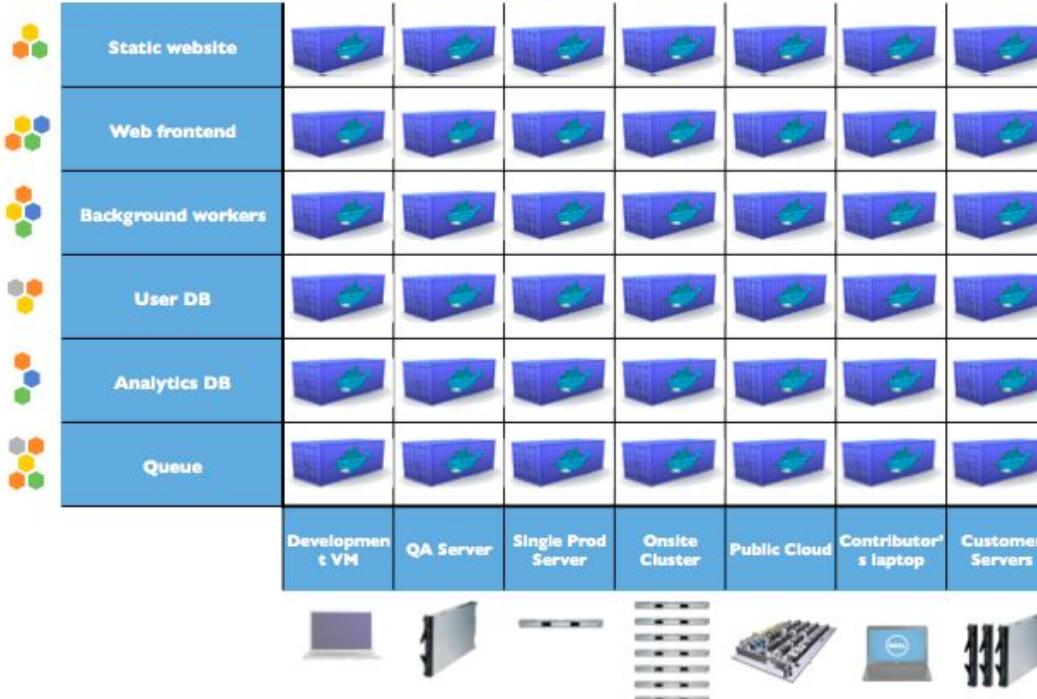
The deployment nightmare



Docker containers



Solving the deployment matrix

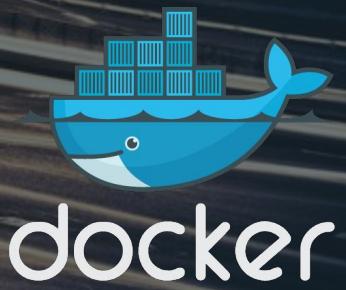


Benefits of Docker

- 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



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 homepage. At the top, there is a navigation bar with links for Dashboard, Explore (which is highlighted with a red box), and Organizations. To the right of the navigation is a search bar with a magnifying glass icon and the word "Search", also highlighted with a red box. Further to the right are "Create" and "trainingteam" dropdown menus. Below the navigation bar, a section titled "Explore Official Repositories" is displayed. It features three repository cards for "centos official", "busybox official", and "ubuntu official". Each card includes a Docker logo icon, the repository name, the number of stars (1.4K, 298, 2.3K respectively), the number of pulls (2.2M, 36.7M, 23.0M respectively), and a "DETAILS" button with a right-pointing arrow.

Repository	Stars	Pulls	Actions
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



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

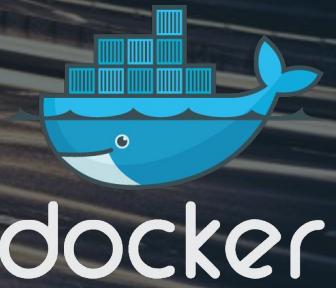
```
docker pull ubuntu
```

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

```
docker pull ubuntu:12.04
```



Building Images



Module objectives

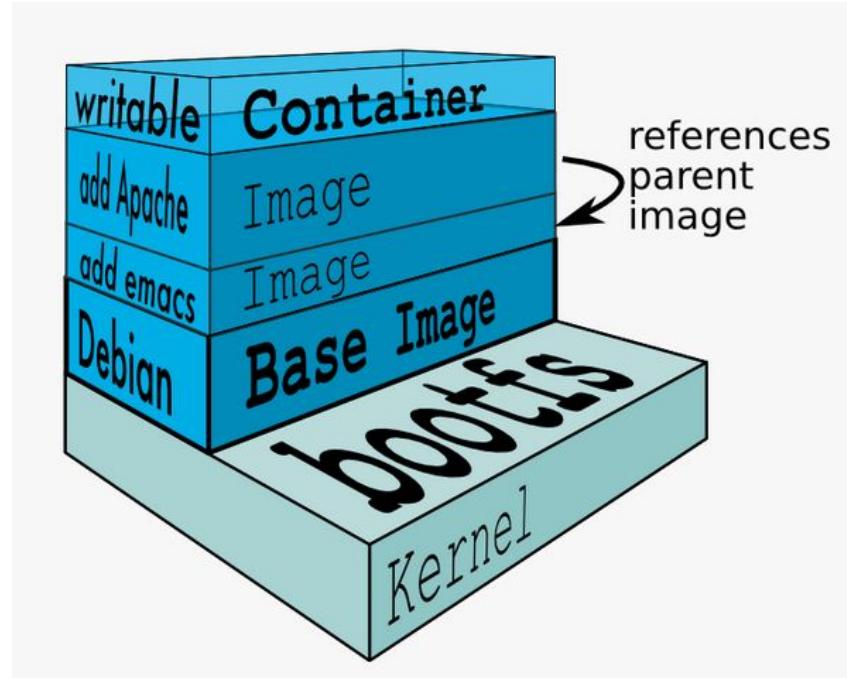
In this module we will

- 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



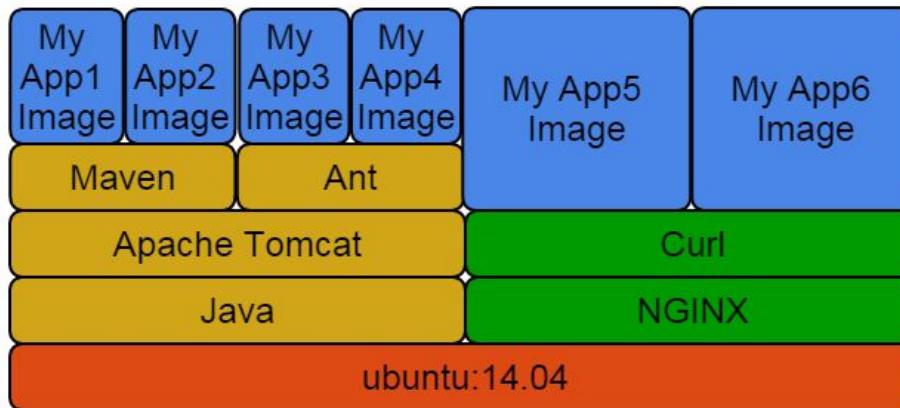
Understanding image layers

- 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



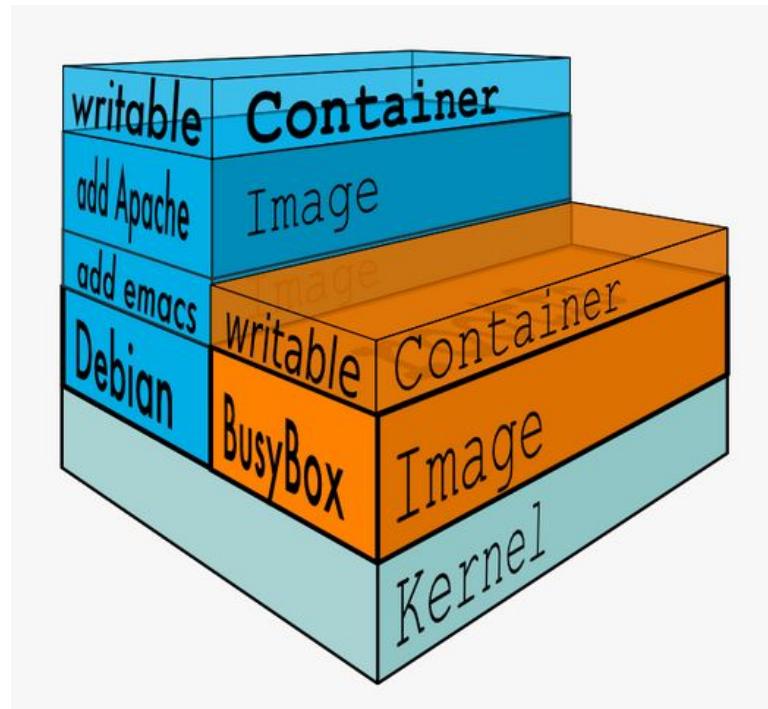
Sharing layers

- 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



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
```



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



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.



A Simple Dockerfile

```
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
```



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
```



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 .`



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



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"]
```



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"]
```



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



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



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



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
```



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"]
```

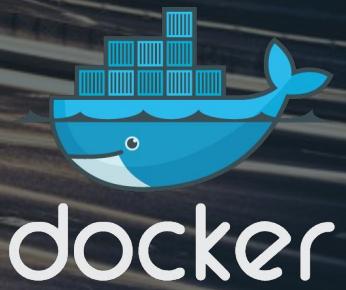


Module summary

- 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



Docker in Continuous Integration



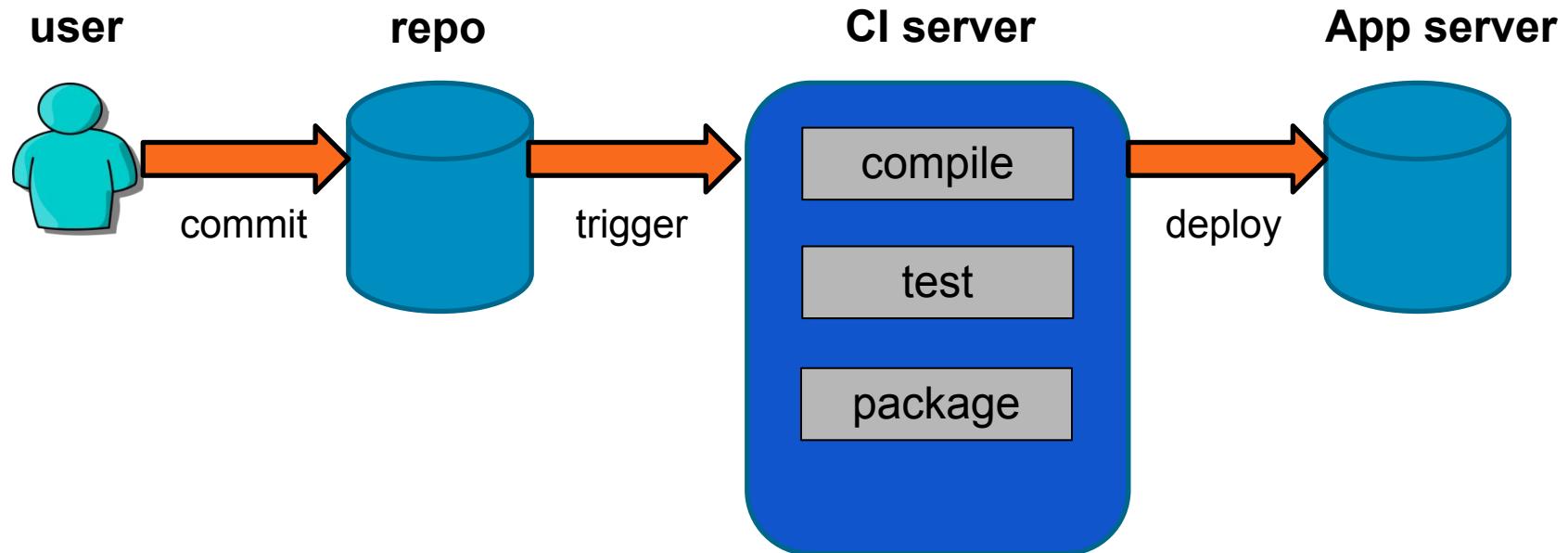
Module objectives

In this module we will:

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

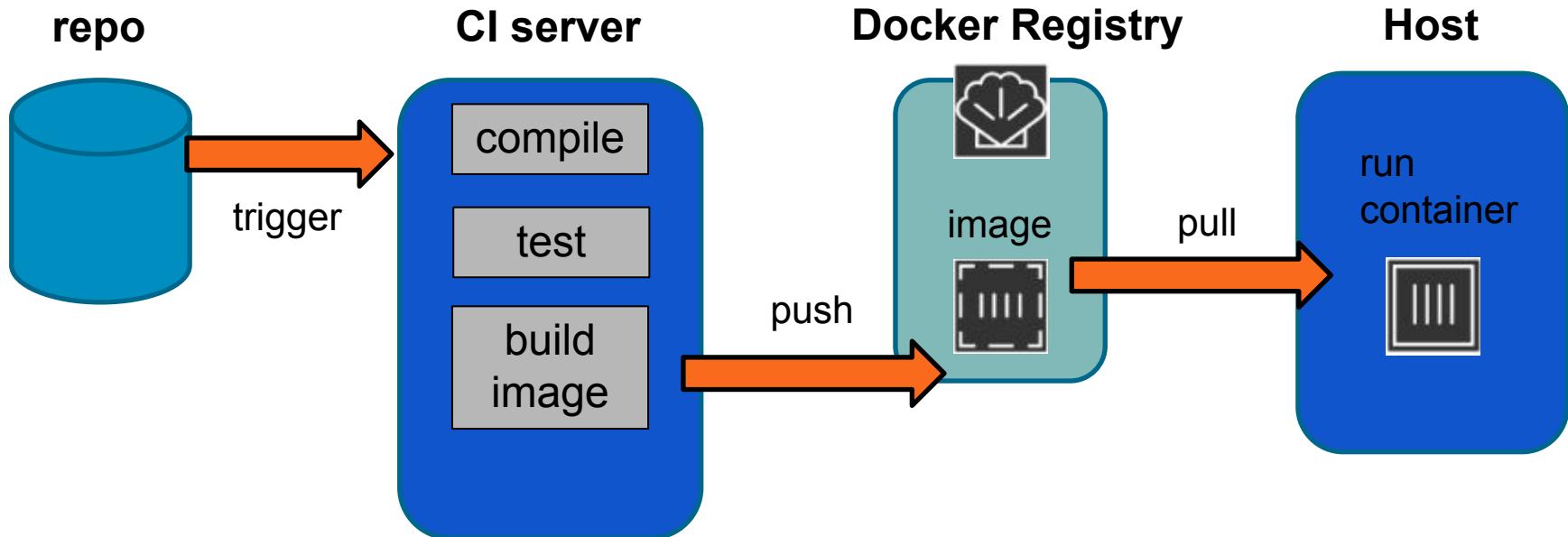


Traditional Continuous Integration



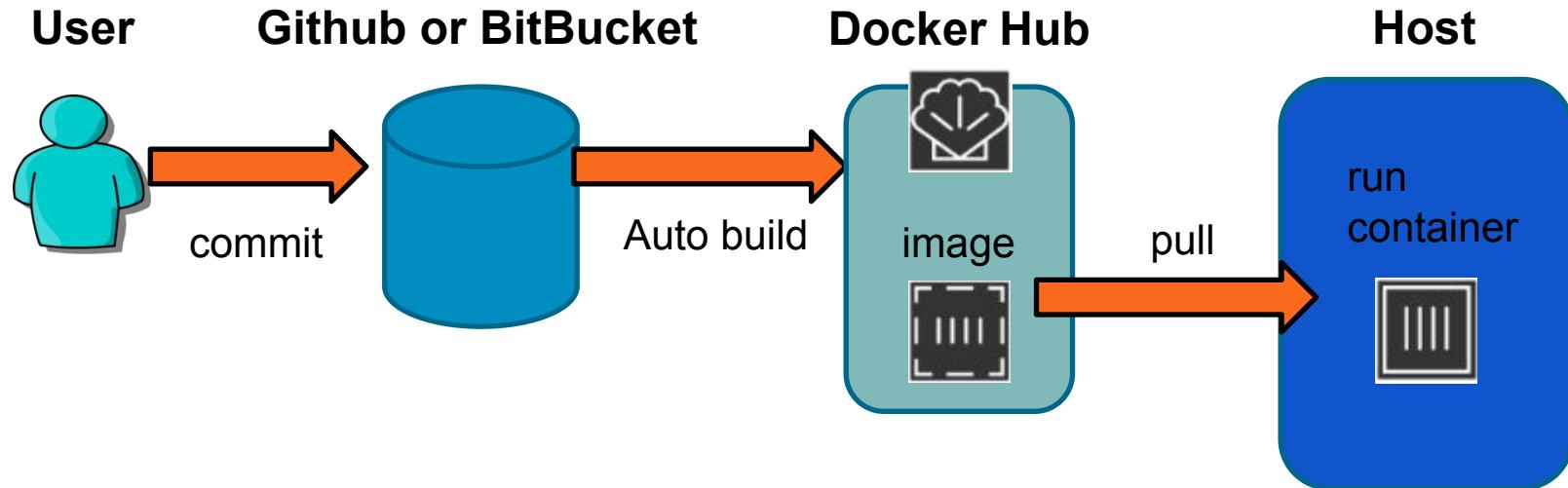
Using Docker in CI

- 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



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

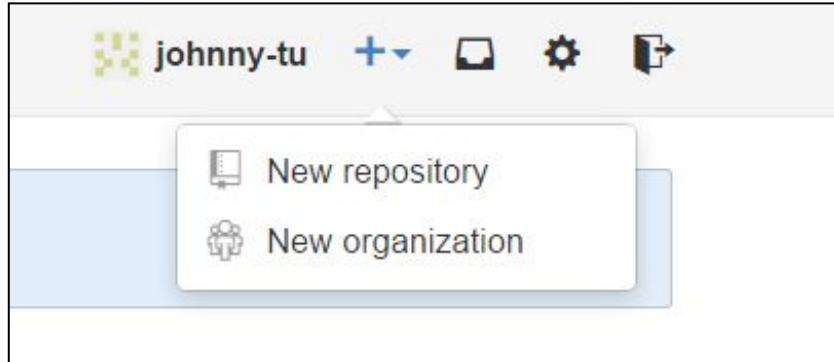
Unlimited collaborators
Unlimited public repositories

- ✓ 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`



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

The screenshot shows a GitHub repository page for 'johnny-tu / javahelloworld'. The page includes a 'Quick setup' section with options for 'Set up in Desktop' (selected), 'HTTPS', and 'SSH'. It also shows the repository URL: <https://github.com/johnny-tu/javahelloworld.git>. Below this, it says 'We recommend every repository include a README, LICENSE, and .gitignore.' A red box highlights the '...or create a new repository on the command line' section, which contains the following git commands:

```
echo # javahelloworld >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/johnny-tu/javahelloworld.git
git push -u origin master
```

Below this, another section says '...or push an existing repository from the command line' with the following git commands:

```
git remote add origin https://github.com/johnny-tu/javahelloworld.git
git push -u origin master
```

The right sidebar shows repository statistics: 1 unwatched, 0 stars, 0 issues, 0 pull requests, 0 wiki pages, 0 pulse events, 0 graphs, and settings.



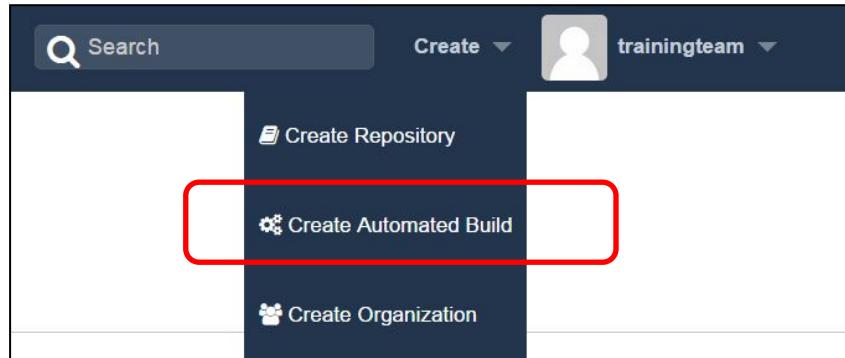
Push our code into the repository

- 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`



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, there are two sections: "Users/Organizations" and a search bar labeled "Type to filter".

Users/Organizations:

- docker-training (represented by a blue ship icon)
- johnny-tu (represented by a green and yellow grid icon)

Type to filter:

- HelloRedis
- 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

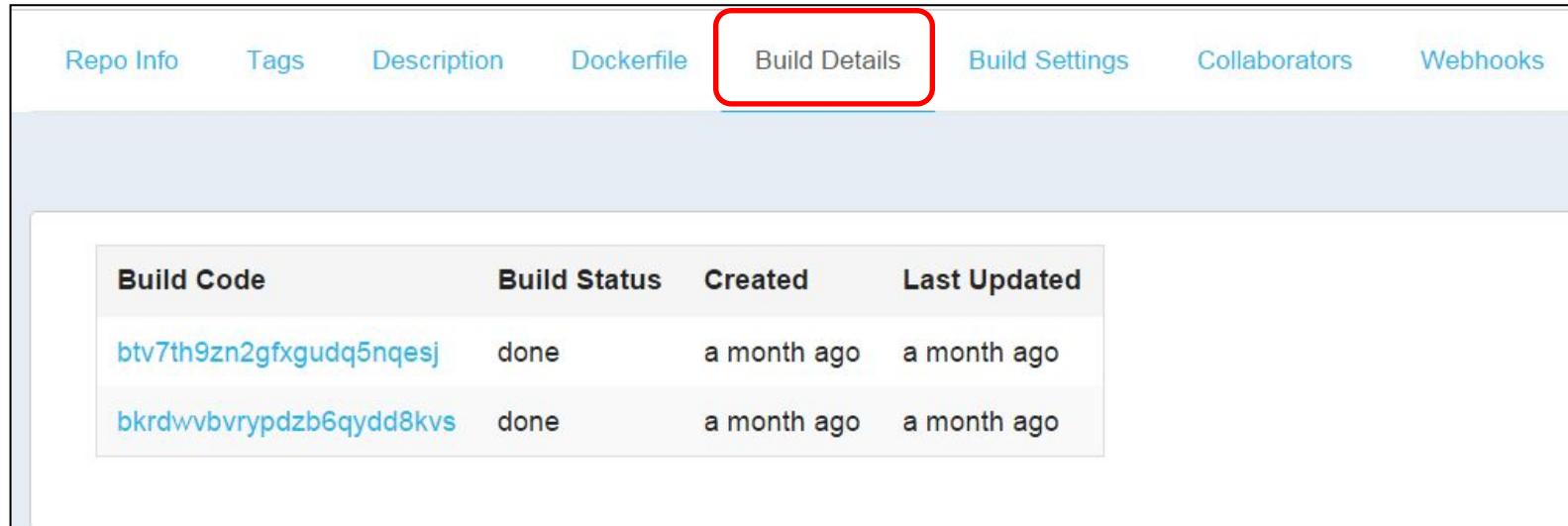
The screenshot shows a configuration page for an automated build. At the top, there are two input fields: a dropdown menu set to "trainingteam" and a text input field containing "javahelloworld". Below these is a "Short Description (100 Characters)" text area. The main configuration section has four columns: "Type", "Name", "Dockerfile Location", and "Tag". Under "Type", a dropdown menu is set to "Branch". Under "Name", the value "master" is entered. Under "Dockerfile Location", the value "/" is entered. Under "Tag", the value "latest" is entered. To the right of the "Tag" column is a blue button with a white plus sign (+), which is likely used to add more build configurations.

Type	Name	Dockerfile Location	Tag
Branch	master	/	latest



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 the 'Build Details' tab highlighted with a red box. Below it, a table displays two completed builds. The columns are: Build Code, Build Status, Created, and Last Updated.

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

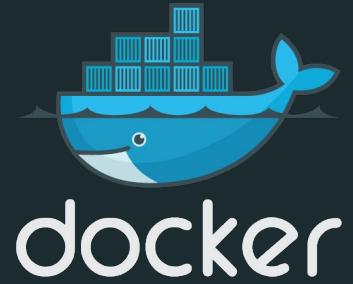


Module summary

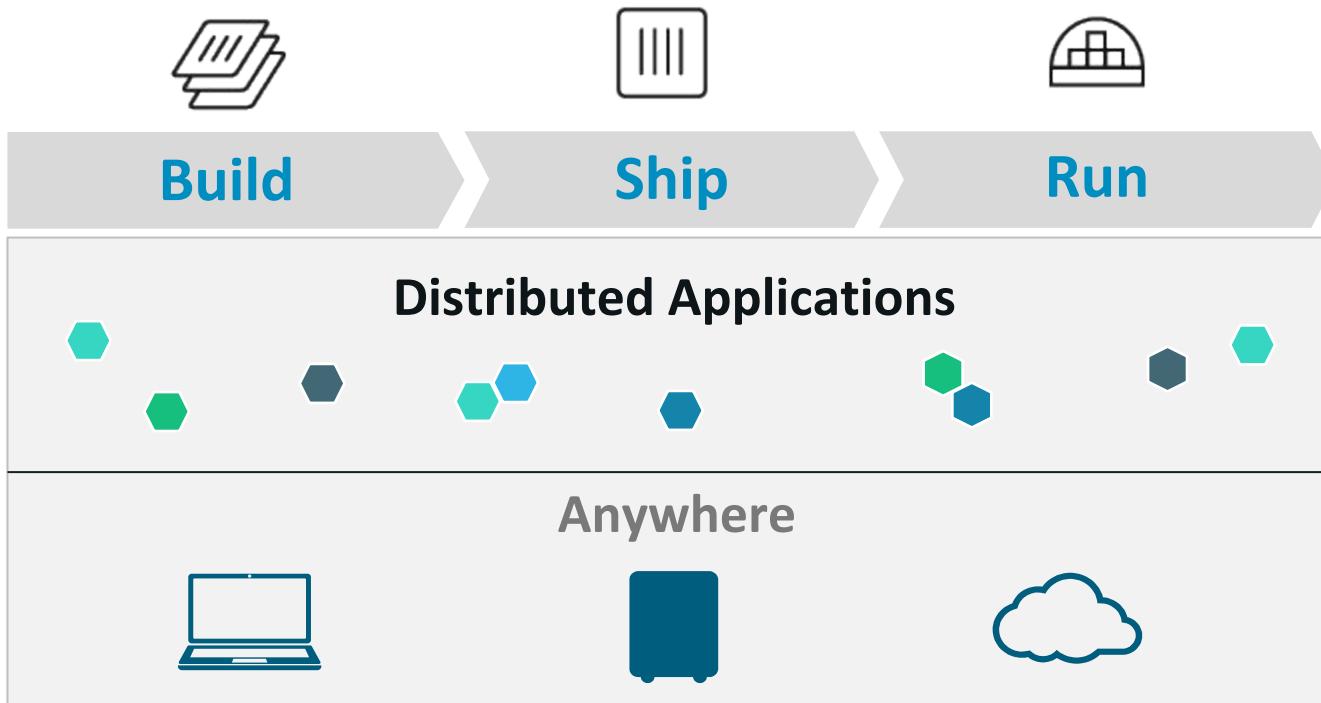
- There are many ways for Docker containers to fit into your continuous integration or continuous delivery process
- Docker Hub's auto build repository is one way for us to build and distribute production ready images



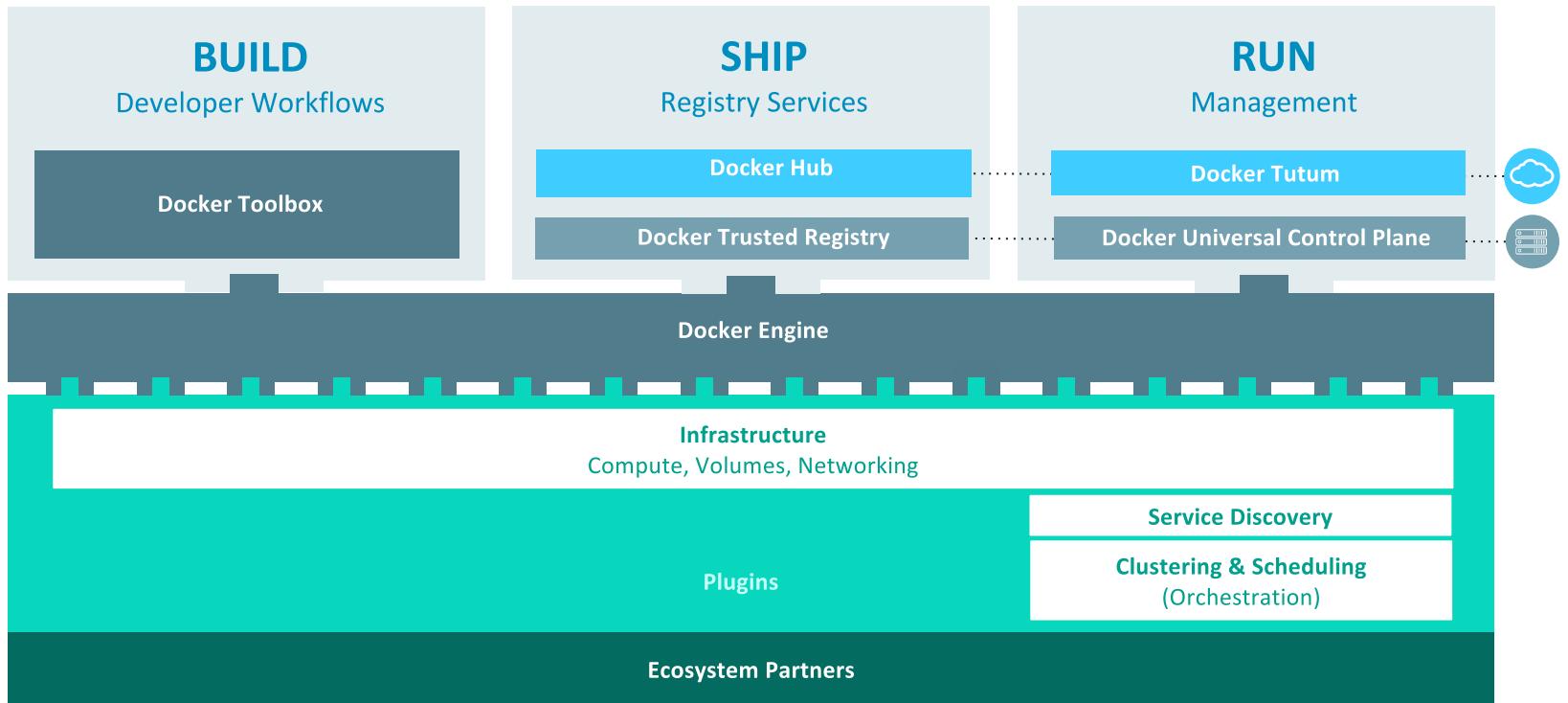
What's New @ Docker



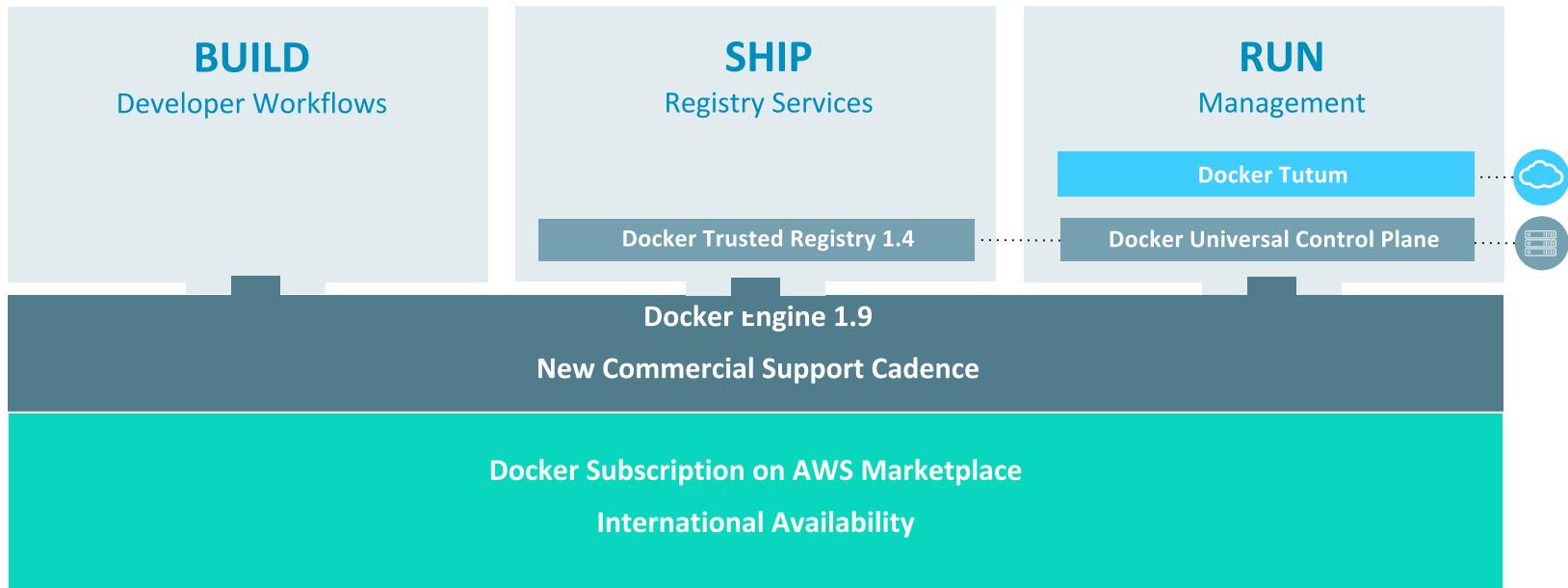
The Docker mission



Docker Containers as a Service platform



What's New in Q4



Docker Subscription: Docker Trusted Registry and Docker Engine



What's New

Docker Trusted Registry 1.4

Docker Engine 1.9

New Commercial Support Cadence





Docker Trusted Registry 1.4

User Experience

- Updated GUI
- Search & browse
- Interactive API documentation

Image Management

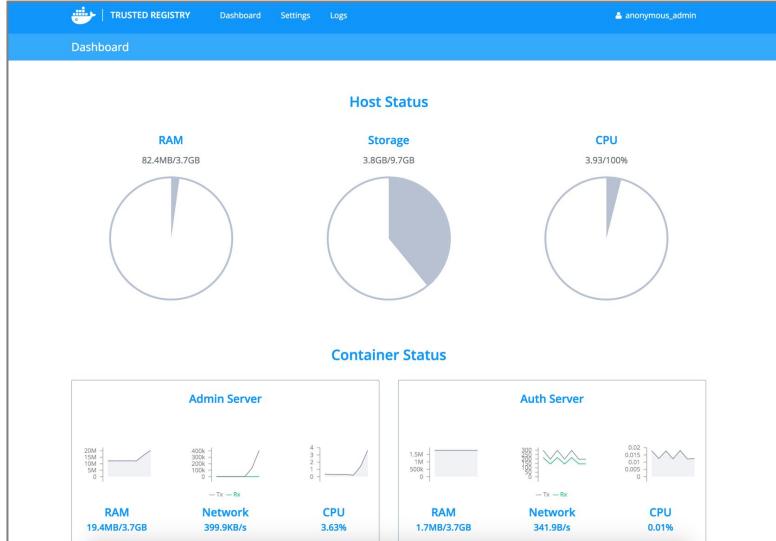
- Manage repos
- Soft delete
- Garbage collection
- Content Trust
(experimental support)

Infrastructure Management

- Registry 2.1.x support
- Engine 1.9 support



Trusted Registry: User Experience



Completely new look for the user interface

The repositories page lists several repositories under the 'All' tab. Each repository entry includes the name, a brief description, and a 'private' badge if applicable. The repositories listed are:

- admin/another-repo: this is a test repo
- admin/i-love-docker
- admin/the-best-repo-ever [private]: shhh this is a private repo
- admin/trusted-registry-repo
- admin/release123: An important repo
- admin/repos-are-cool [private]
- admin/repo-name-here: This is an optional description of this repo!

Search and browse image repos.
Make repos public or private.
Assign access by user or group.



Trusted Registry: Image Management

Settings

General Security Storage License Auth Garbage collection Updates

Trusted Registry can run garbage collection on your storage backend to remove deleted tags and images.

Schedule

Cron schedule for garbage collection, such as '0 0 0 * *' to run daily at midnight. A cron expression represents a set of times in the series: seconds, minutes, hours, day of month, month, day of week.

0 0 0 * *

Save

Last garbage collection run information

Garbage collection has not yet ran.

Delete image tags from the repo and then schedule garbage collection to optimize storage

Notary Server (experimental feature)

Notary server url. Note that for Notary signatures to show up in the DTR UI you must use the same domain name when pushing as the domain name configured in DTR. Ex. <https://172.17.42.1:4443>

Notary Verify TLS (experimental feature)

Whether or not to verify that the TLS certificate is valid for the Notary server. This is necessary for production environments.

Notary TLS Root CA (experimental feature)

The TLS certificate of the Certificate Authority used to verify Notary's certificate (if not already in operating system's CA store).

TLS certificate

Enable Content Trust image signing in Trusted Registry *





Docker Engine 1.9

Networking

- Docker native networking
- Ecosystem Plugins

Volume Management

- Persistent storage
- Ecosystem Plugins

Monitoring

- AWS CloudWatch driver
- Disc I/O metrics in Stats API





Docker Engine 1.9: Networking

Problem: Distributed applications services require a distributed way to communicate with each other

Solution: Transform networking as Docker transformed compute



Developers

- Consistent Docker experience
- App defined, multi-host networking



IT Operations

- Portable across environments
- Plugin support from ecosystem



vmware





Docker Engine 1.9: Volume Management

Problem: Difficult to manage persistent storage for ephemeral containers

Solution: First class volume support enables flexible persistent storage



Developers

- Consistent Docker experience
- Full Docker API support
- Easier to setup



IT Operations

- Easier to manage
- Portable across environments
- Flexible plugin support from ecosystem



Flocker[®]
by ClusterHQ



EMC {code}



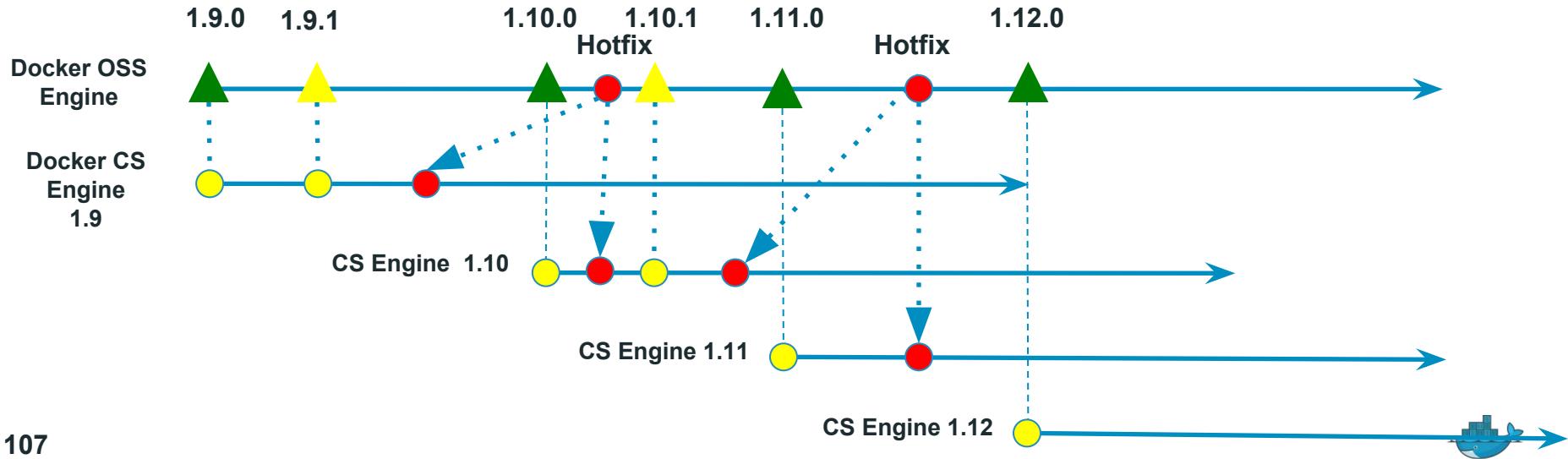
■■■■■ **Blockbridge**



New Commercial Support Cadence

Docker Engine

- Three major releases supported in parallel
- Support time period is 6 months per release or period for 3 releases
- Patches and hotfixes are back ported to all supported versions



Subscription Plans

Server

Docker Trusted Registry

Docker Engines

Commercial Support
(12 or 24 hour)

Cloud

Docker Hub

Docker Engines

Commercial Support
(12 or 24 hour)



Introducing Tutum

The best way to deploy and manage Docker in production



Tutum: The best way to deploy and manage Dockerized distributed apps in production



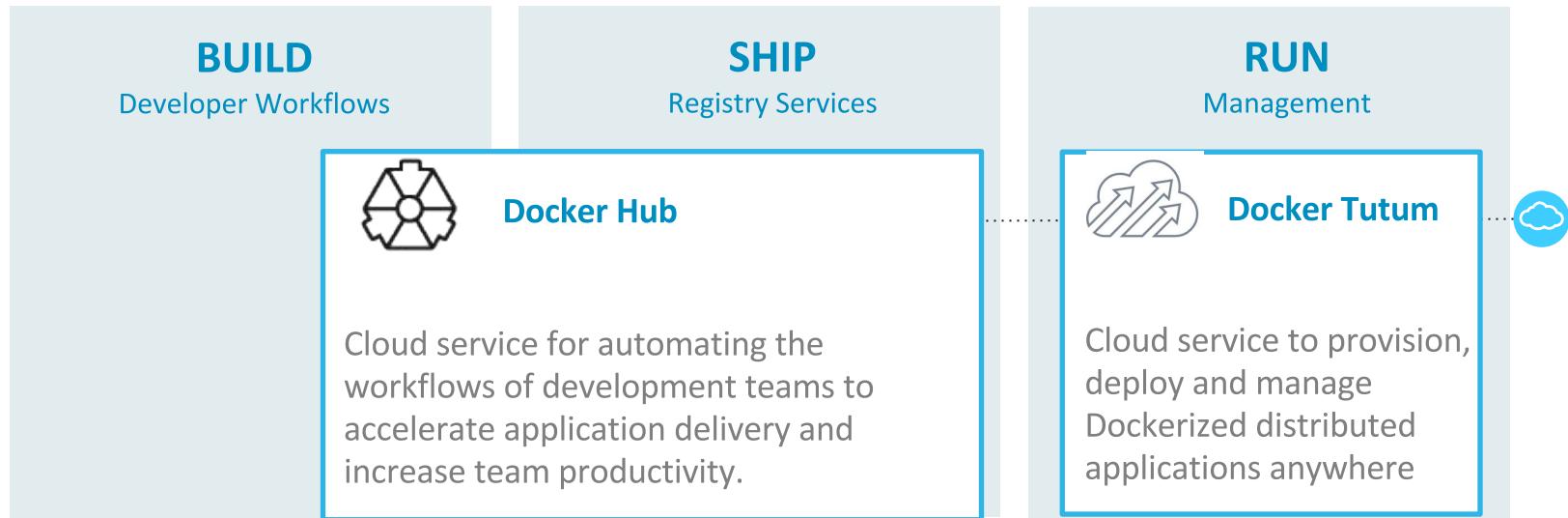
For Devs: Enables DevOps initiatives and self service for deploying and managing distributed apps

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

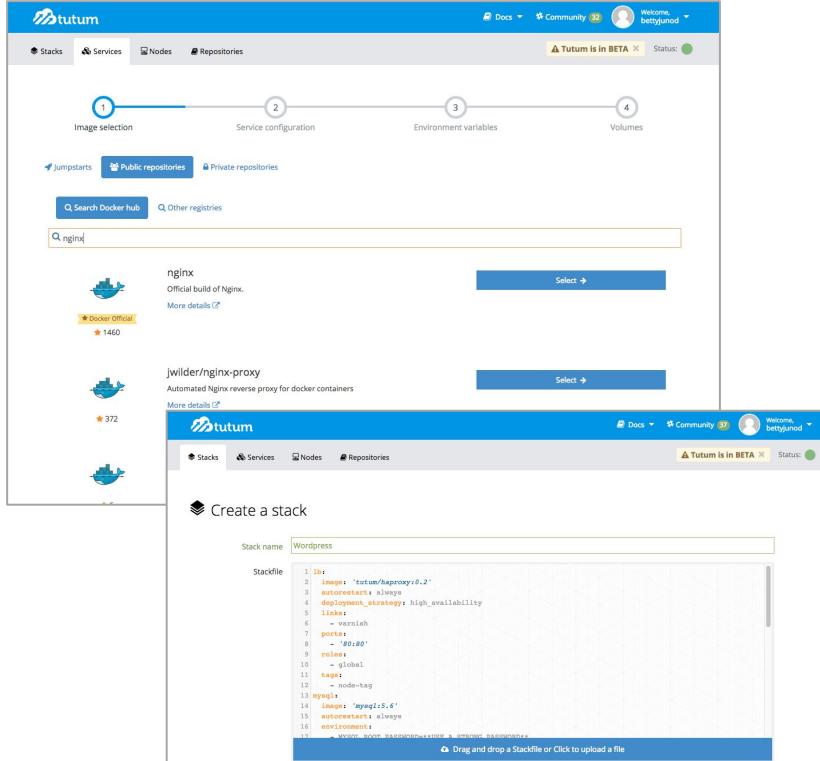
One Platform: Purpose built on Docker for Docker to bring together dev and ops workflows



Tutum + Docker Hub complete the distributed app lifecycle



Deploy and manage distributed apps



- Integrates to existing CI/CD workflows
- Integrates with Docker Hub to provide a complete commercial build, ship, run solution
- Point and click GUI, CLI and RESTful API experience
- Easily create simple single service or complex multi-service applications



Manage distributed apps anywhere

The screenshot displays a service management interface with three main sections:

- Service dashboard:** Shows a list of services with columns for Name, Status, and Image. Services listed include `unibench-5d123d5d` (Running), `postgres-18736db` (Starting), `nginx-m2d7392` (Running), `web-green` (Running), `web-blue` (Running), and `lb` (Running). Each service card includes actions like Stop, Terminate, and Redeploy.
- unixbench-5d123d5d:** A detailed view of a running service. It shows a timeline from 03:00 to 21:00 on Oct 14, 2015. A CPU usage chart shows spikes between 10% and 100%. Below the chart, there are configuration options for `cpu.shares`, `cpu.nice`, `mem.limit_in_bytes`, and `mem.reservation_in_bytes`.
- hello-world-74c70d8d:** A view of a service that is not running. It shows a configuration section with `hello` and `hello-world-test` configurations, both set to Off. Below this is a history of recent actions:
 - Service Start (13:07 10/01/2015)
 - Service Start (13:01 10/01/2015)
 - Service Create (13:01 10/01/2015)Each action includes a timestamp, duration, location (San Francisco, United States), IP (116.23.20.130), user agent (Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2434.101 Safari/537.36), request (POST /api/v1/service/), and request body ({"name": "hello-world-74c70d8d", "container_envars": {}, "linked_to_service": null}). A message at the bottom indicates the service creation was successful.

- Intuitive dashboards provide holistic visibility
- Gain insight with logs, monitoring and service history
- One click create, start, terminate, deploy and redeploy
- Point and click to change configurations and scale containers



Deploy and manage anywhere

Account info

Cloud Providers

</> Source Providers

API key

Change password

Change email

Notifications

Connected services

Newsletters

Cancel account

Link a provider to start deploying nodes.
Provider not listed? That's okay! Use our [Bring your own node feature](#).

Provider	Account	Status	Action
Amazon Web Services	(no account linked)	✗	+ Add credentials
Digital Ocean	(no account linked)	✗	+ Link account
Microsoft Azure	(no account linked)	✗	+ Add credentials
SoftLayer	(no account linked)	✗	+ Add credentials
Packet	(no account linked)	✗	+ Add credentials

Node dashboard

Launch new node cluster + Bring your own node

Actions

Filter by node clusters	
Test-Node 0	⋮
Test-Node 0	⋮
Map_Test 0	⋮
Map_2 0	⋮
MapTest 0	⋮
Test 0	⋮
Spacecat-Nodes... 0	⋮
Spacecat-Nodes 0	⋮
Spacecat-Nodes 2	⋮
cloudcat-azure 0	⋮
cloudcat-azure 0	⋮
cloudcat-azure 2	⋮
cloudcat-aws 0	⋮
cloudcat-packet 0	⋮
cloudcat-aws 1	⋮
unibench 0	⋮
unibench 1	⋮

Node	Status	Node cluster	Action
c224f830-manomarks.node.tutum.io	Deployed	Spacecat-Nodes	Deploy Terminate
✗ 1 Container % 173.192.139.3 SoftLayer Seattle ✗ 1 x 2.0 GHz Core - 1 GB Docker 1.7.1			
c045d8d3-manomarks.node.tutum.io	Deployed	Spacecat-Nodes	Deploy Terminate
✗ 1 Container % 173.192.139.4 SoftLayer Seattle ✗ 1 x 2.0 GHz Core - 1 GB Docker 1.7.1			
09eb20a-manomarks.node.tutum.io	Deployed	cloudcat-azure	Deploy Terminate
✗ 1 Container % 40.78.158.14 Microsoft Azure Central US ✗ Basic AO Docker 1.7.1			
68835706-manomarks.node.tutum.io	Deployed	cloudcat-azure	Deploy Terminate
✗ 1 Container % 40.122.130.215 Microsoft Azure Central US ✗ Basic AO Docker 1.7.1			
0d132003-manomarks.node.tutum.io	Deployed	cloudcat-packet	Deploy Terminate
✗ 1 Container % 147.75.194.65 Packet Parsippany, NJ ✗ Ubuntu 14.04 LTS - Type 1 Docker 1.7.1			
b6f0902d-manomarks.node.tutum.io	Deployed	cloudcat-aws	Deploy Terminate
✗ 1 Container % 52.24.197.127 Amazon Web Services us-west-2b ✗ 12.micro Docker 1.7.1			
91292d55-manomarks.node.tutum.io	Deployed	unibench	Deploy Terminate
✗ 0 Containers % 52.89.110.228 Amazon Web Services us-west-2b ✗ 12.micro Docker 1.7.1			

- Infrastructure flexibility for dev and ops teams
- Securely link to any cloud provider
- Bring your own private infrastructure
- One step to provision, install configure and cluster Docker Engines
- Point and click to scale and manage your Docker environment



Docker Subscription for AWS

www.docker.com/aws



Docker Subscription for AWS



A screenshot of the Docker Trusted Registry for AWS product page in the AWS Marketplace. The page shows the Docker logo, a brief description of the service, and various configuration options like customer rating, latest version, and AWS services required. It also includes sections for highlights, pricing details, and software pricing options.

116

www.docker.com/aws

Streamline Deployment

- Available in AWS Marketplace (US and Europe)
- Business Day Support AMIs
 - Integrated EC2 + Docker costs per hour
 - 30 day free trial for Docker
 - Trusted Registry and Engine AMIs
- Bring Your Own License (BYOL)
 - Trusted Registry AMI available
 - Engines need to be manually installed

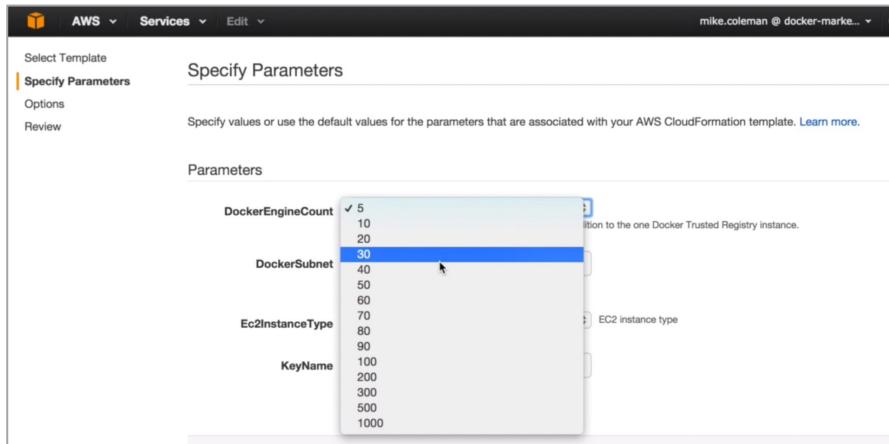


Docker Cloud Formation Template



Accelerate time to value

- Automate Deployment
 - EC2, Trusted Registry and CS Engines with Business Day support
 - Point and click to configure deployment
- 30 day free Docker trial



International Availability

Docker Subscription available for Europe



Hourly and annual
subscriptions available
from AWS Marketplace



Subscription licenses
available

L1 and L2 support provider
for US and Europe



Bring your own license to
deploy Docker VHD in
Azure Marketplace to
European zones



Next Steps

Questions & Answers

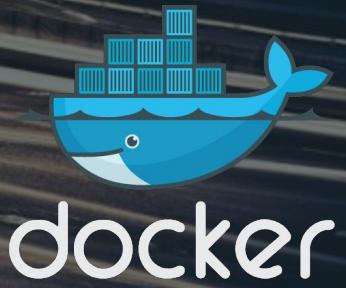
Try Docker on AWS free for 30 days

- Streamlined eval and deploy
- Try NEW features in Trusted Registry 1.4 and Engine 1.9

Sign up for Tutum Beta



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 (**New in 1.9!!**)

- 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
```



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



(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"
  }
]
```



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



(cont'd)

1. Create another file called test2.txt

```
touch test2.txt
```

2. Exit the superuser account

```
exit
```

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

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

4. Change directory into the /www/website folder

5. 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



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  
  },  
,
```



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

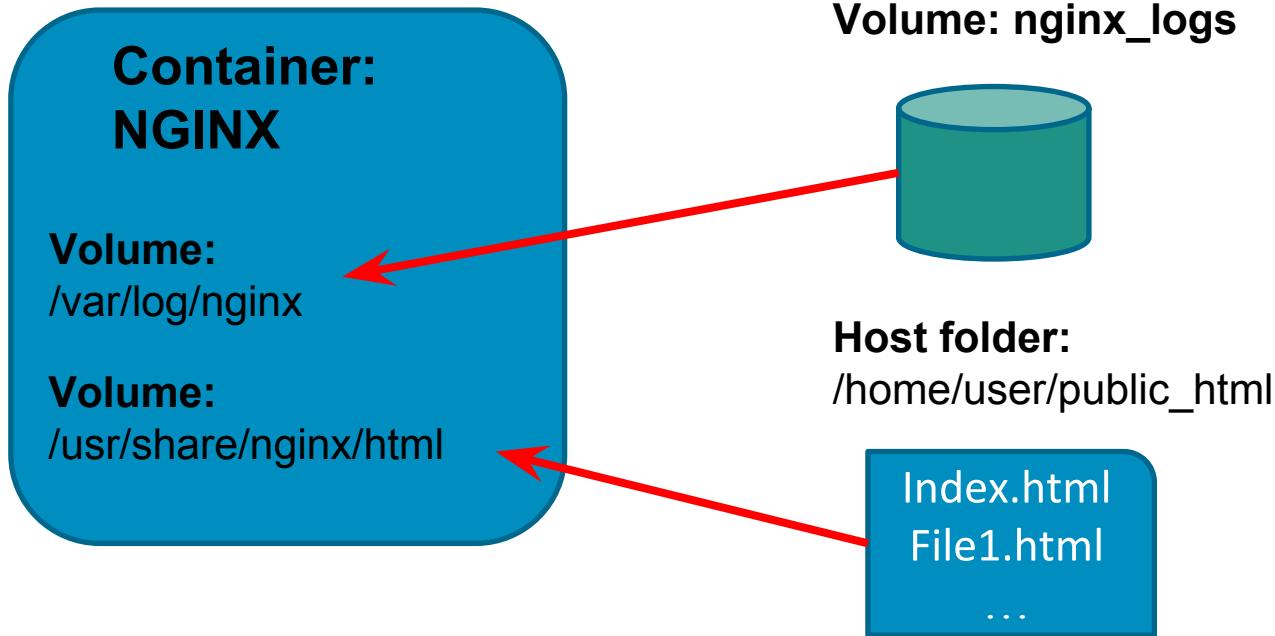


A data persistence example with NGINX

- 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
```



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
```



(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



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`)



(cont'd)

1. Elevate user privileges

```
sudo su
```

2. Change directory into the volume path

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

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

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

5. 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

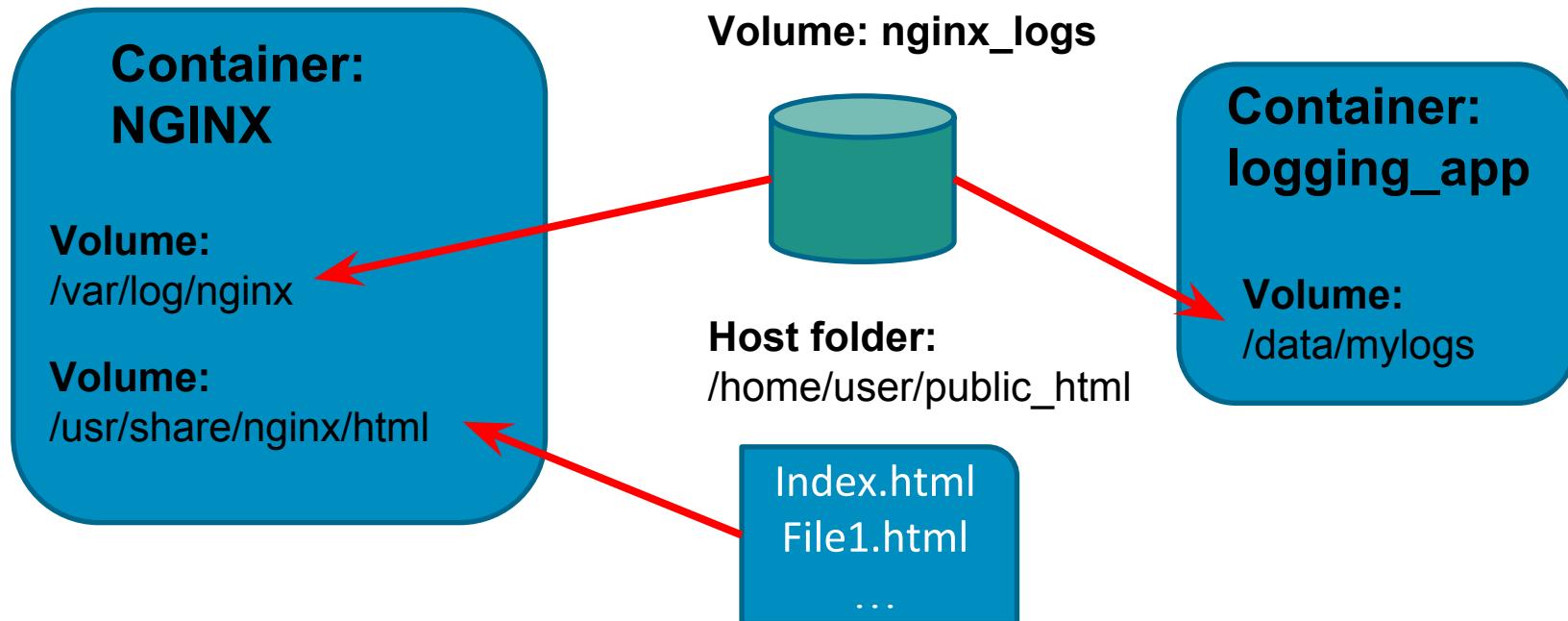


Sharing data between containers

- 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



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/website1.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"]
```



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



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#
```



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

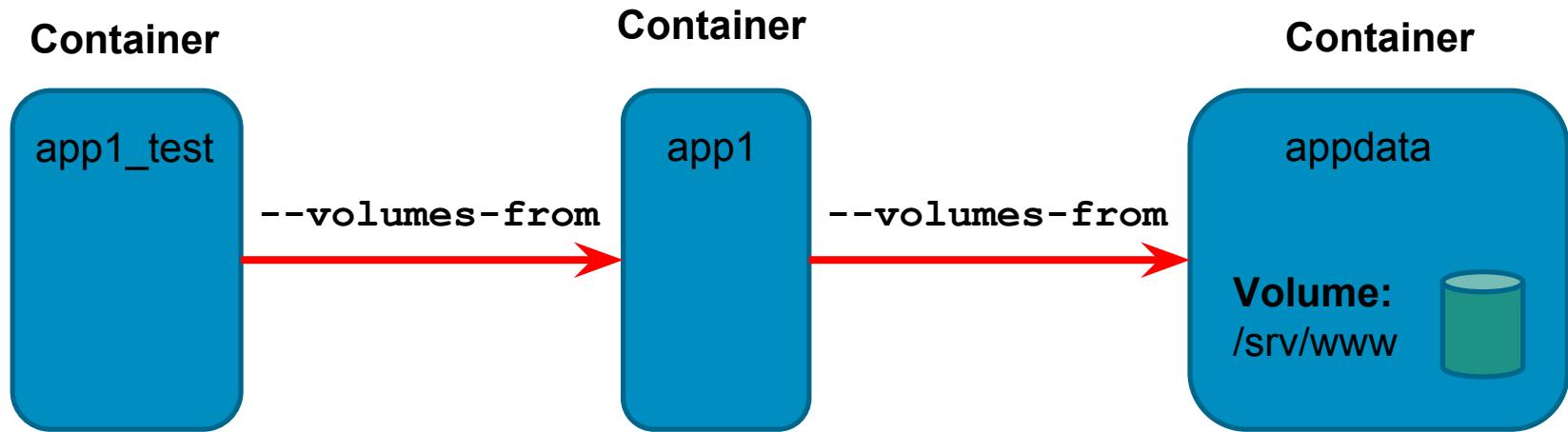
```
docker run -it --name webserver --volumes-from webdata  
ubuntu:14.04
```

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`



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 \  
  
```



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
```



Backup our webserver volume

1. Create a folder called `backups` in your home directory
2. Backup the volume in your webserver container with the following commands

```
> cd && mkdir backups  
> docker run -ti --rm --volumes-from webserver -v  
$(pwd) /backups:/backups ubuntu:14.04  
> tar cvf /backups/webserver.tar /srv/www  
> exit  
> cd backups
```

Check your backup folder for the tar file and verify contents

```
> tar -tvf webserver.tar
```



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}},
  {}]}]
```

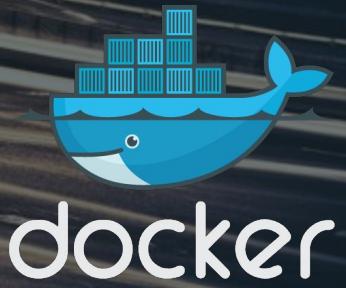


Module summary

- 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



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



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:#[
```



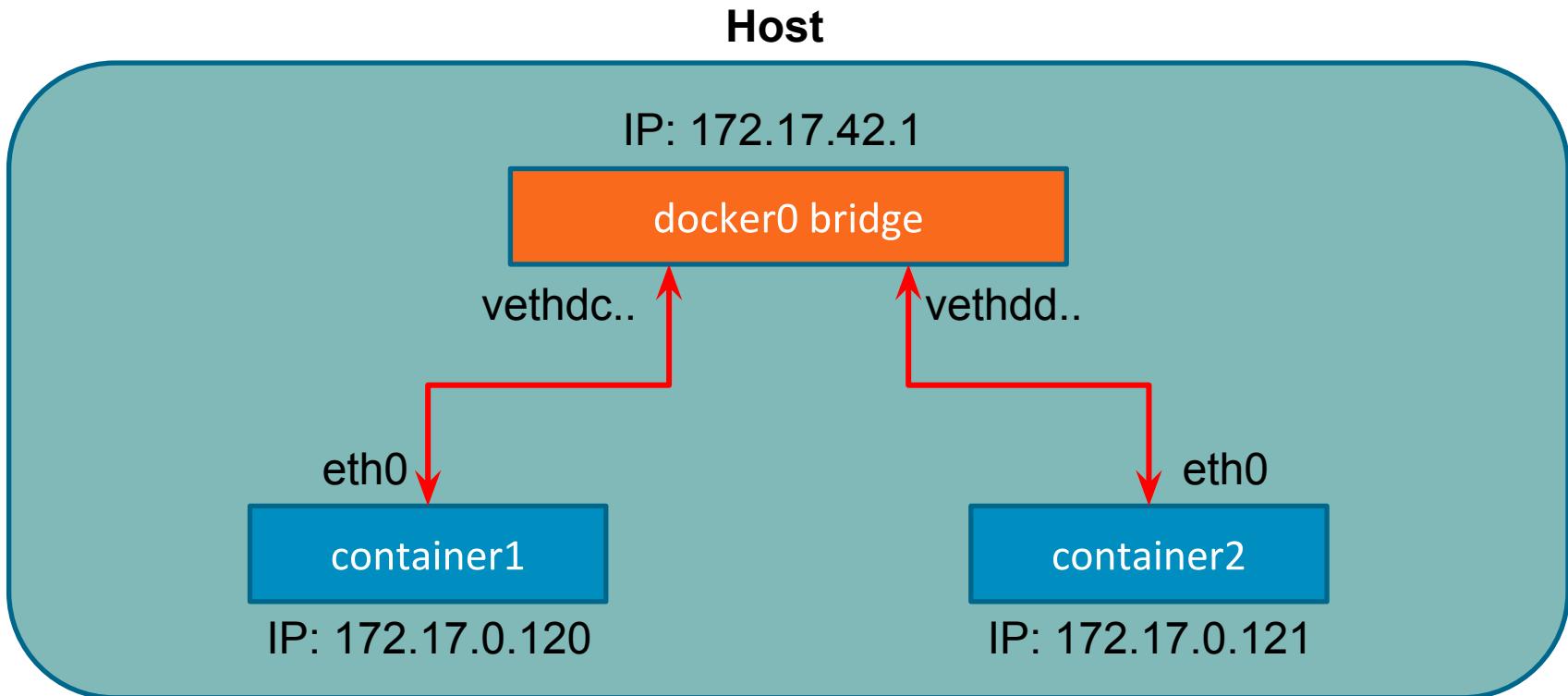
Check container networking properties

- 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(**New in 1.9!!**)

- 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 **local** 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": ""
      }
    }
  }
]
```



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

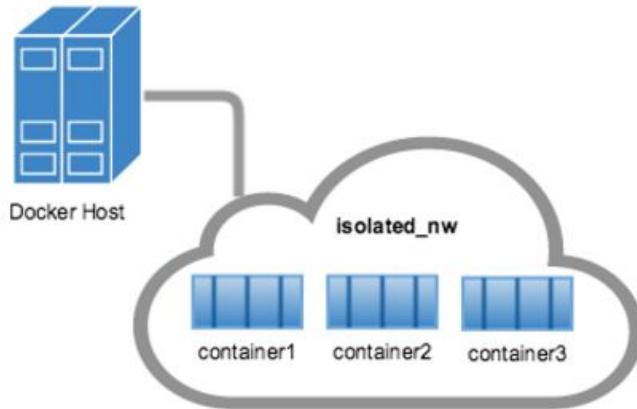
- Inspect the container and look for the Networks field in the docker inspect output
- Should indicate the Network name, Gateway, IP Address etc...

```
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



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



(cont'd)

6. Check for the following entry

172.18.0.2 ubuntu3

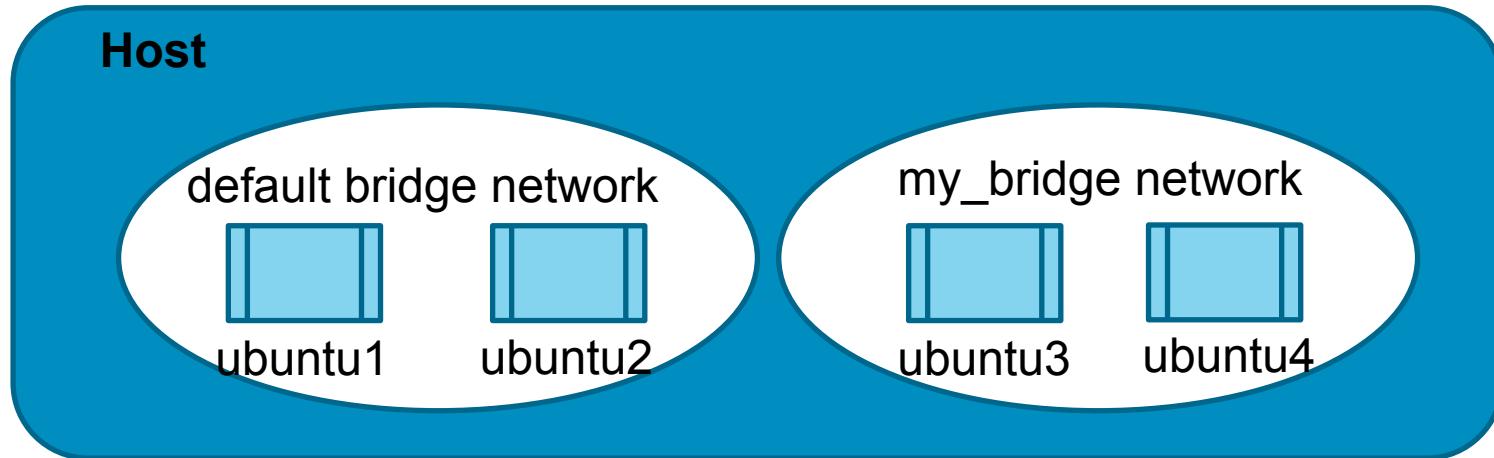
172.18.0.2 ubuntu3.my_bridge

7. Ping `ubuntu3` using the container name.
8. Try to ping the `ubuntul` 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"  
    }  
}
```



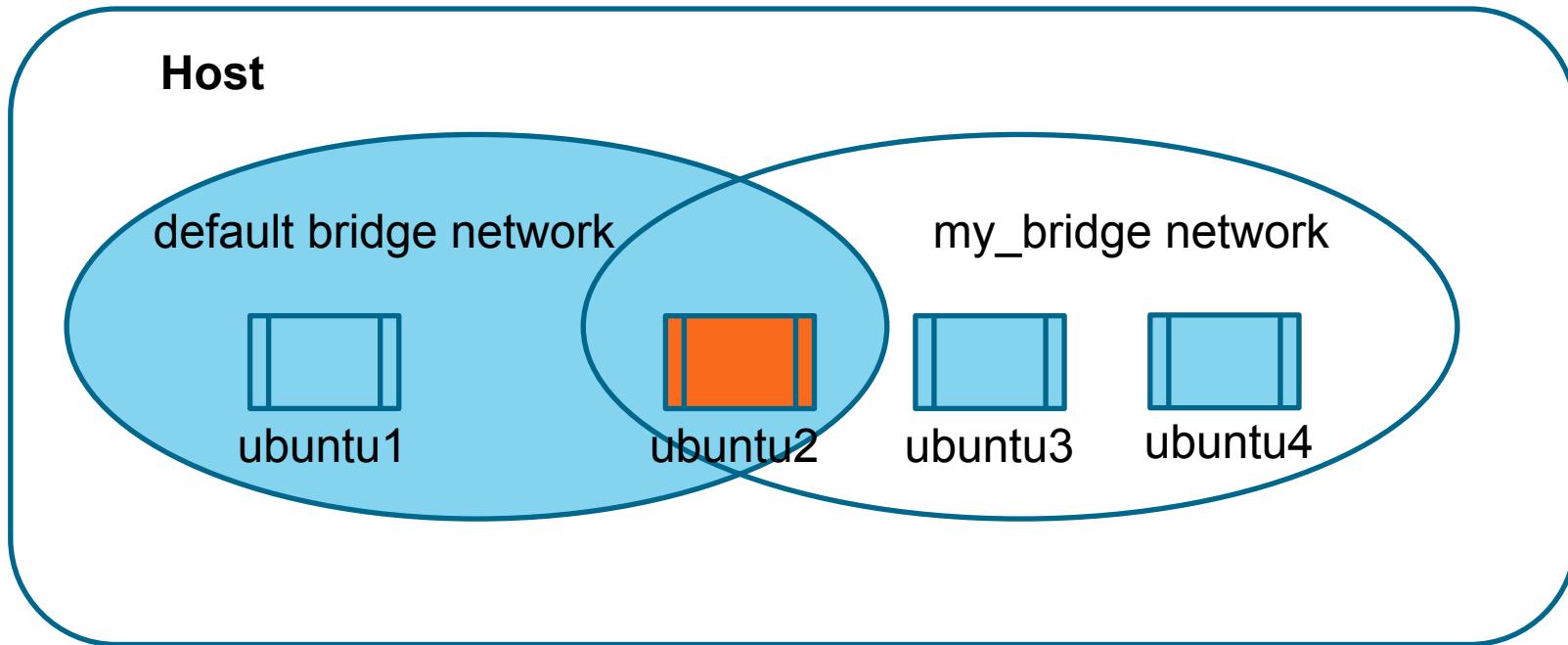
Checking the /etc/hosts file

- 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



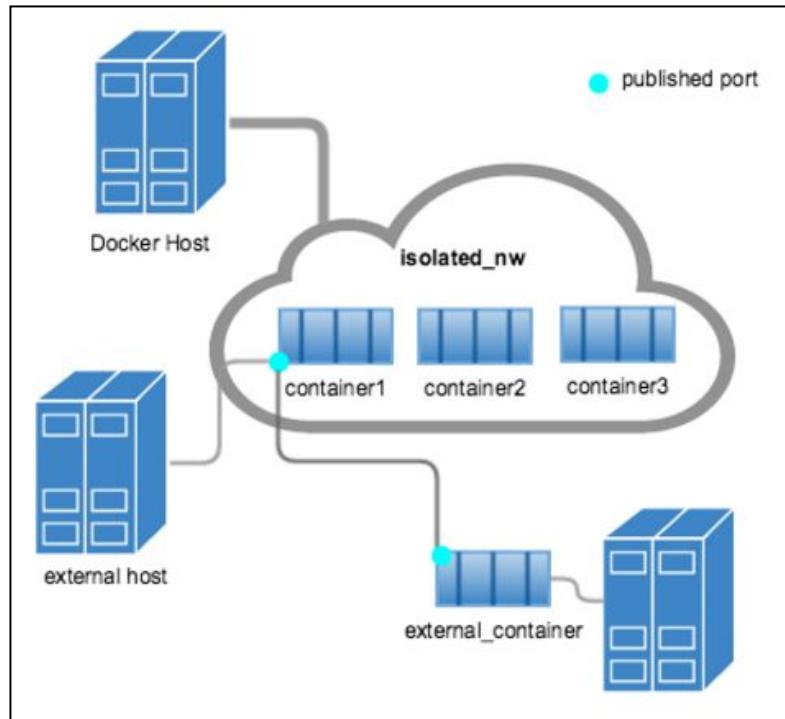
Connect to multiple networks

1. Connect your `ubuntu2` container to the `my_bridge` network
`docker network connect my_bridge ubuntu2`
2. Check the `/etc/hosts` file on `ubuntu2` and confirm that you can see the `ubuntu3` and `ubuntu4` containers
`docker exec ubuntu2 cat /etc/hosts`
3. Check the `/etc/hosts` file on `ubuntu3` and confirm that you can see the `ubuntu2` and `ubuntu4` containers
`docker exec ubuntu3 cat /etc/hosts`
4. Ping the `ubuntu2` container from `ubuntu3`
`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 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
```



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;"]
```



Multi-host networking

- Containers running on different hosts cannot communicate with each other without mapping their TCP ports to the host's TCP ports
- Multi-host networking allows these containers to communicate without requiring port mapping
- The Docker Engine supports multi host networking natively out of the box via the overlay network driver
- Requirements for creating an overlay network
 - Access to a key-value store
 - A cluster of hosts connected to the key-value store
 - All hosts must have Kernel version 3.16 or higher
 - Docker Engine properly configured on each host



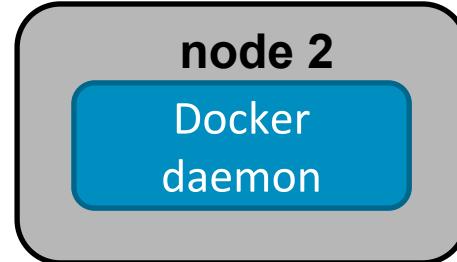
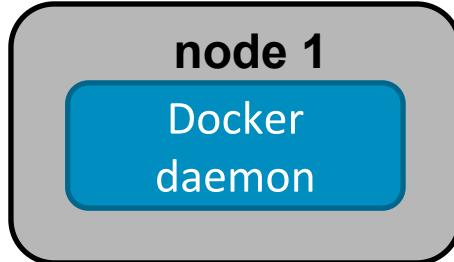
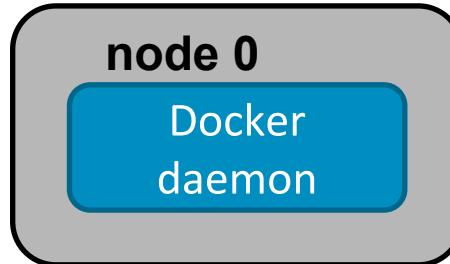
Key-value store

- Stores information about the network state including
 - Discovery
 - Endpoints
 - IP addresses
- Supported options
 - Consul
 - Zookeeping (Distributed store)
 - Etcd
 - BoltDB (Local store)



Our current setup

- You have three AWS instances with one designated as master and then two as nodes
- We will setup the key-value store on the node 0.



Step 1 - Setup key-value store

Perform this on your node-0

- We will use Consul as our key-value store
- Run consul in a container with the following command
docker run -d -p 8500:8500 -h consul --name consul \
program/consul -server -bootstrap
- Check that consul is running and that port 8500 is mapped to the host

```
student@dockerhost:~$ docker run -d -p 8500:8500 -h consul --name consul program/consul -server -bootstrap  
2c54ff68658d937aee4161735c65de43aef71ba31cdc24645161e65d8d3aa175  
student@dockerhost:~$ docker ps  
CONTAINER ID        IMAGE               ...      STATUS            PORTS          NAMES  
2c54ff68658d        program/consul    ...      Up 1 seconds     .... 0.0.0.0:8500->8500/tcp   consul  
student@dockerhost:~$
```



Step 2 – Configure Docker Engines

- The Docker Engine on Node1 and Node2 needs to be configured to:
 - Listen on TCP port 2375
 - Use the Consul key-value store on our node-0 created in step 1
- **Note:** You will need to know the private IP address of your node 0

Switch over to Node 1

- To configure the Docker daemon, open the /etc/default/docker file

```
sudo vim /etc/default/docker
```



Step 2 – Configure Docker Engines

- Modify the DOCKER_OPTS variable that what is shown on the bottom of this slide.
 - Replace <node-0 IP> with the IP address of your node-0.
 - Should be done on one line
- Save your changes

```
DOCKER_OPTS="-H tcp://0.0.0.0:2375 \
-H unix:///var/run/docker.sock \
--cluster-store=consul://<node-0 IP>:8500/network \
--cluster-advertise=eth0:2375"
```



Step 2 – Configure Docker Engines

- Once the `/etc/default/docker` file has been configured, restart your Docker daemon
`sudo service docker restart`
- Verify that Docker is running.
 - You can run `docker ps` and make sure there is output
- Repeat the same process on Node 2**



Step 3 – Configure the Overlay network

Perform on either Node1 or Node2

- We will create an overlay network called multinet
- It will be configured with the 10.10.10.0/24 subnet
- Run the command

```
docker network create -d overlay --subnet 10.10.10.0/24  
multinet
```

```
root@node1:~$ docker network create -d overlay --subnet 10.10.10.0/24 multinet  
91107e4f66395df21b4d35520ffe282bbfa80bde9ec3b129f8c3f00ae65bea36  
root@node1:~$ docker network ls  
NETWORK ID      NAME      DRIVER  
91107e4f6639    multinet  overlay  
73e6a15d82a8    none      null  
7cf377412587    host      host  
9108538181e4    bridge    bridge
```

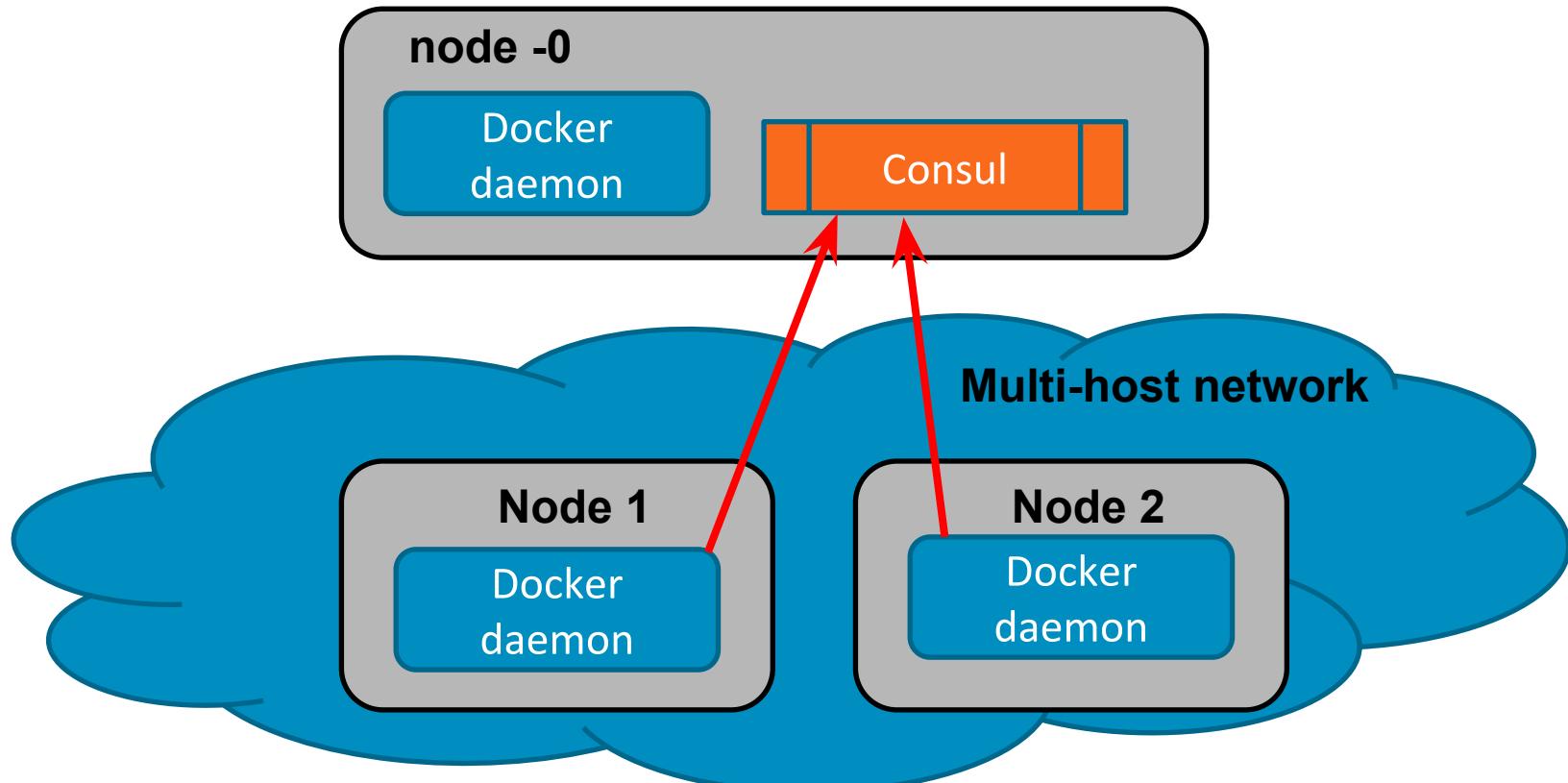


Step 3 – Configure the Overlay network

- Once you have created the overlay network, check that it is present
`docker network ls`
- Switch to your other Node and check that the network is present as well



Our updated setup



Setup the multi-host network

1. Make sure you have no containers running on any of your nodes
2. Make sure you have the IP address of your node 0.
3. Follow steps 1, 2 and 3 on the previous slides to create your multi-host network



Running containers on a multi-host network

- To run a container on the multi-host network, you just need to specify the network name on the `docker run` command. For example:
`docker run -itd --name c1 --net multinet busybox`
- Can run containers from any host connected to the network
- Container will be assigned an IP address from the subnet of your multi-host network
- The first time an overlay network is created on any host, Docker also creates another network called `docker_gwbridge`.
- The `docker_gwbridge` network provides external access for containers
- All TCP/UDP ports are open on an overlay network and thus, it is not necessary to map container ports to host ports in order for containers to communicate



Checking container network config

```
root@node1a:~$ docker run -itd --name c1 --net multinet busybox  
4fb56a1e8128e62f3e48ea1aad66fa68f532b63c5d811223246321de30e89979\  
root@node1a:~$ docker exec c1 ifconfig  
eth0      Link encap:Ethernet HWaddr 02:42:0A:0A:0A:02  
          inet addr:10.10.10.2 Bcast:0.0.0.0 Mask:255.255.255.0  
          inet6 addr: fe80::42:aff:fe0a:a02/64 Scope:Link  
            UP BROADCAST RUNNING MULTICAST MTU:1450 Metric:1  
            RX packets:15 errors:0 dropped:0 overruns:0 frame:0  
            TX packets:8 errors:0 dropped:0 overruns:0 carrier:0  
            collisions:0 txqueuelen:0  
            RX bytes:1206 (1.1 KiB) TX bytes:648 (648.0 B)  
eth1      Link encap:Ethernet HWaddr 02:42:AC:12:00:02  
          inet addr:172.18.0.2 Bcast:0.0.0.0 Mask:255.255.0.0  
          inet6 addr: fe80::42:acff:fe12:2/64 Scope:Link  
            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1  
            RX packets:8 errors:0 dropped:0 overruns:0 frame:0  
            TX packets:8 errors:0 dropped:0 overruns:0 carrier:0  
            collisions:0 txqueuelen:0  
            RX bytes:648 (648.0 B) TX bytes:648 (648.0 B)  
lo        Link encap:Local Loopback  
          ....
```

Multi-host network with subnet of 10.10.10.0/24

docker_gwbridge network



Container discovery

- When a container is added to a multi-host overlay network, the /etc/hosts file will contain entries for all other containers
- Docker will also automatically update the /etc/hosts file of all other containers and add an entry for the new container
- Containers must be run with a defined name (i.e. use the --name option)
- Containers with randomly generated names will not be “discovered”



Container discovery

- At this stage, we have setup a container on Node1 called c1
- Let's run another container on Node2 and check the /etc/hosts file

```
root@node2:~# docker run -d --name nginx --net multinet nginx
E7576fd798aac025bc73c395011adc73007fe184c1646c645ed99afacffc177b
root@node2:~# docker exec nginx cat /etc/hosts
10.10.10.3      e7576fd798aa
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
10.10.10.2      c1
10.10.10.2      c1.multinet
```



Entry for container c1 on host node1



Run container on multi-host network

Using Node1

1. Run an ubuntu container on your multi-host network called c1
docker run -itd --name c1 --net multinet ubuntu:14.04
2. Check the network configuration of c1 and verify that you can see an eth0 and eth1 network
docker exec c1 ifconfig

Using Node2

3. Run an NGINX container on your multi-host network called nginx. Do not specify any port mapping
docker run -d --name nginx --net multinet nginx
4. Check the /etc/hosts file and verify that you can see a host entry for your c1 container
docker exec nginx cat /etc/hosts



(cont'd)

5. Ping your nginx container using your nginx container. Do not use the container IP address in the ping command
`docker exec c1 ping nginx`

Using Node1

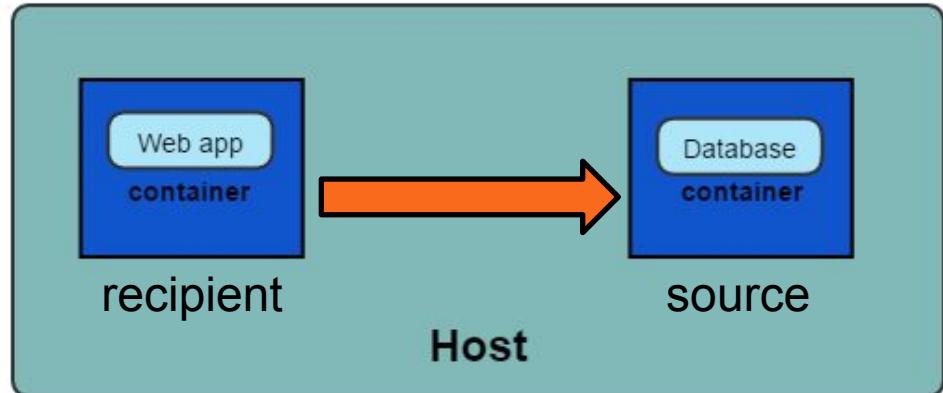
6. Check the /etc/hosts file and verify there is an entry for the nginx container
`docker exec c1 cat /etc/hosts`
7. Get terminal access into c1
`docker exec -it c1 bash`
8. Install curl
`apt-get install curl`
9. Use curl to make a request to the NGINX server running on Node2. You should notice that we can request for the NGINX welcome page despite our NGINX server running in a container on a different hosts without any port mapping
`curl nginx`



Linking Containers (same-host)

Linking is a communication method between containers which allows them to securely transfer data from one to another

- Source and recipient containers
- Recipient containers have access to data on source containers
- Links are established based on container names



Uses of Linking

- Containers can talk to each other without having to expose ports to the host
- Essential for micro service application architecture
- Example:
 - Container with Tomcat running
 - Container with MySQL running
 - Application on Tomcat needs to connect to MySQL



Creating a Link

1. Create the source container first
 2. Create the recipient container and use the `--link` option
- **Best practice –** give your containers meaningful names
 - Format for linking
`name:alias`

Create the source container using the postgres

```
docker run -d --name database postgres
```

Create the recipient container and link it

```
docker run -d -P --name website --link database:db nginx
```



The underlying mechanism

- Linking provides a secure tunnel between the containers
- Docker will create a set of environment variables based on your --link parameter
- Docker also exposes the environment variables from the source container.
 - Only the variables created by Docker are exposed
 - Variables are prefixed by the link alias
 - ENV instruction in the container Dockerfile
 - Variables defined during `docker run`
- DNS lookup entry will be added to `/etc/hosts` file based on your alias



Link two Containers

1. Run a container in detached mode using the tomcat image.

Name the container “appserver”

```
docker run -d --name appserver tomcat
```

2. Run another container using the Ubuntu image and link it with the “appserver” container. Use the alias “appserver”, run the bash terminal as the main process

```
docker run -it --name client \
    --link appserver:appserver \
    ubuntu:14.04 bash
```

3. In the “client” container terminal, open the /etc/hosts file

4. What can you observe?

5. Ping your Tomcat container without using the IP address

```
ping appserver
```



Environment variables in source container

- Let's inspect the variables defined in our "appserver" Tomcat container

```
$ docker exec appserver env  
PATH=/usr/local/tomcat/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  
HOSTNAME=91794033c488  
LANG=C.UTF-8  
JAVA_VERSION=7u79  
JAVA_DEBIAN_VERSION=7u79-2.5.5-1~deb8u1  
CATALINA_HOME=/usr/local/tomcat  
TOMCAT_MAJOR=8  
TOMCAT_VERSION=8.0.22  
TOMCAT_TGZ_URL=https://www.apache.org/dist/tomcat/tomcat-8/v8.0.22/bin/apache-tomcat-8.0.22.tar.gz  
HOME=/root
```



Environment variables in recipient container

- Now we will check the environment variables in the Ubuntu container which is linked to our “appserver” container

```
$ docker exec client env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=0a17932c340d
TERM=xterm
APP SERVER _PORT=tcp://172.17.0.133:8080
APP SERVER _PORT_8080_TCP=tcp://172.17.0.133:8080
APP SERVER _PORT_8080_TCP_ADDR=172.17.0.133
APP SERVER _PORT_8080_TCP_PORT=8080
APP SERVER _PORT_8080_TCP_PROTO=tcp
APP SERVER _NAME=/client/appserver
APP SERVER _ENV_LANG=C.UTF-8
APP SERVER _ENV_JAVA_VERSION=7u79
APP SERVER _ENV_JAVA_DEBIAN_VERSION=7u79-2.5.5-1~deb8u1
APP SERVER _ENV_CATALINA_HOME=/usr/local/tomcat
APP SERVER _ENV_TOMCAT_MAJOR=8
APP SERVER _ENV_TOMCAT_VERSION=8.0.22
APP SERVER _ENV_TOMCAT_TGZ_URL=https://www.apache.org/dist/tomcat/tomcat-8/v8.0.22/bin/apache-tomcat-8.0.22.tar.gz
HOME=/root
```

Connection information created from the --link option

Created from source container

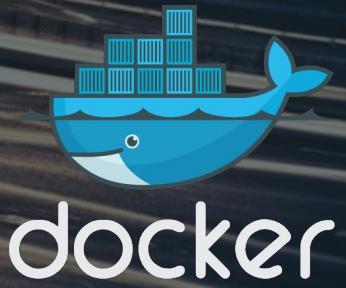


Module summary

- 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
- Containers running on different hosts can communicate with each other without requiring any port mapping if they are part of the same multi-host overlay network
- Key Dockerfile instructions
 - EXPOSE



Docker Content Trust



Module objectives

- Understand how content trust works with Docker images
- Be able to sign your images
- Learn how to configure Docker to use signed images



Docker Content Trust

- Docker Content Trust allows us to ensure the integrity and publisher of Docker images
- Client side signing and verification of image tags can be enforced
- Image publishers sign their images
- Image consumers can ensure their images are signed
- Integrates The Update Framework (TUF) into Docker using Notary
 - <http://theupdateframework.com/>
 - <https://github.com/docker/notary>
- At this stage Docker Content Trust only works for Docker Hub images

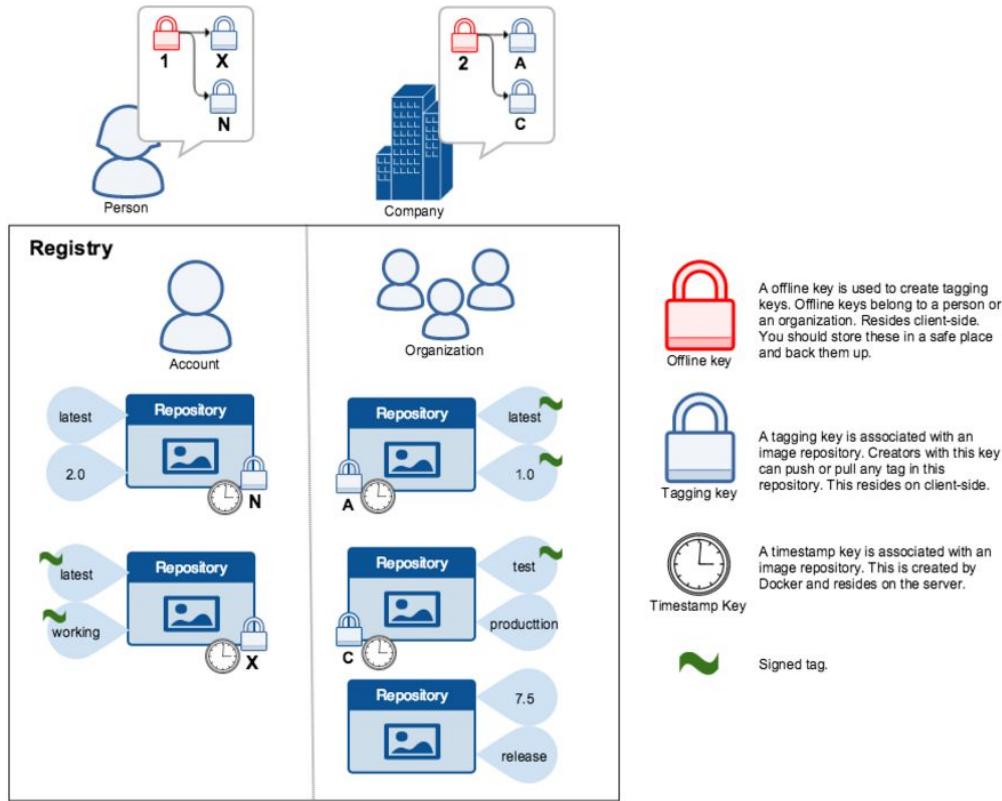


How it works for content publishers

- Content trust is associated with the tag of an image
- Trust for an image tag is managed through the use of signing keys
- Four different keys are used:
 - Root key (also known as offline key)
 - Target and Snapshot key (also known as repository key and tagging key)
 - Timestamp key
- When pushing images to a repository the image is signed with the tagging key
- Different repositories can use the same offline key



Signing keys diagram



Enabling Content Trust

- Docker Content Trust is not enabled by default
- Two ways to enable
 - Set the `DOCKER_CONTENT_TRUST` environment variable to “1”
 - Use the `--disable-content-trust=false` option on an applicable command
- Content trust is applied to the `push`, `pull`, `build`, `create` and `run` commands

Enable content trust at the shell level

```
export DOCKER_CONTENT_TRUST=1
```



Pushing a signed image

- You need to make sure the image is tagged
- The first time a tagged image is pushed with Content Trust enabled, you will be prompted to:
 - Specify a passphrase for the root key
 - Specify a passphrase for your repository key
- The root and repository keys are stored in the `~/.docker/trust` directory

Push an image to Docker Hub and enable content trust to ensure we can sign the image.

```
docker push --disable-content-trust=false jtu/myimage:1.0
```



Pushing a signed image

```
student@masterhost:~$ docker push trainingteam/trustedubuntu:1.0
The push refers to a repository [docker.io/trainingteam/trustedubuntu] (len:
1)
1d073211c498: Image already exists
5a4526e952f0: Image already exists
99fcaefe76ef: Image already exists
c63fb41c2213: Image already exists
1.0: digest: sha256:
bc428b9e892de428cd9e4274b499b0198e1f92dff5a591f370425325088a624e size: 7748
Signing and pushing trust metadata
Enter key passphrase for root key with id 309d07d:
Enter passphrase for new repository key with id docker.
io/trainingteam/trustedubuntu (e815b57):
Repeat passphrase for new repository key with id docker.
io/trainingteam/trustedubuntu (e815b57):
Finished initializing "docker.io/trainingteam/trustedubuntu"
```

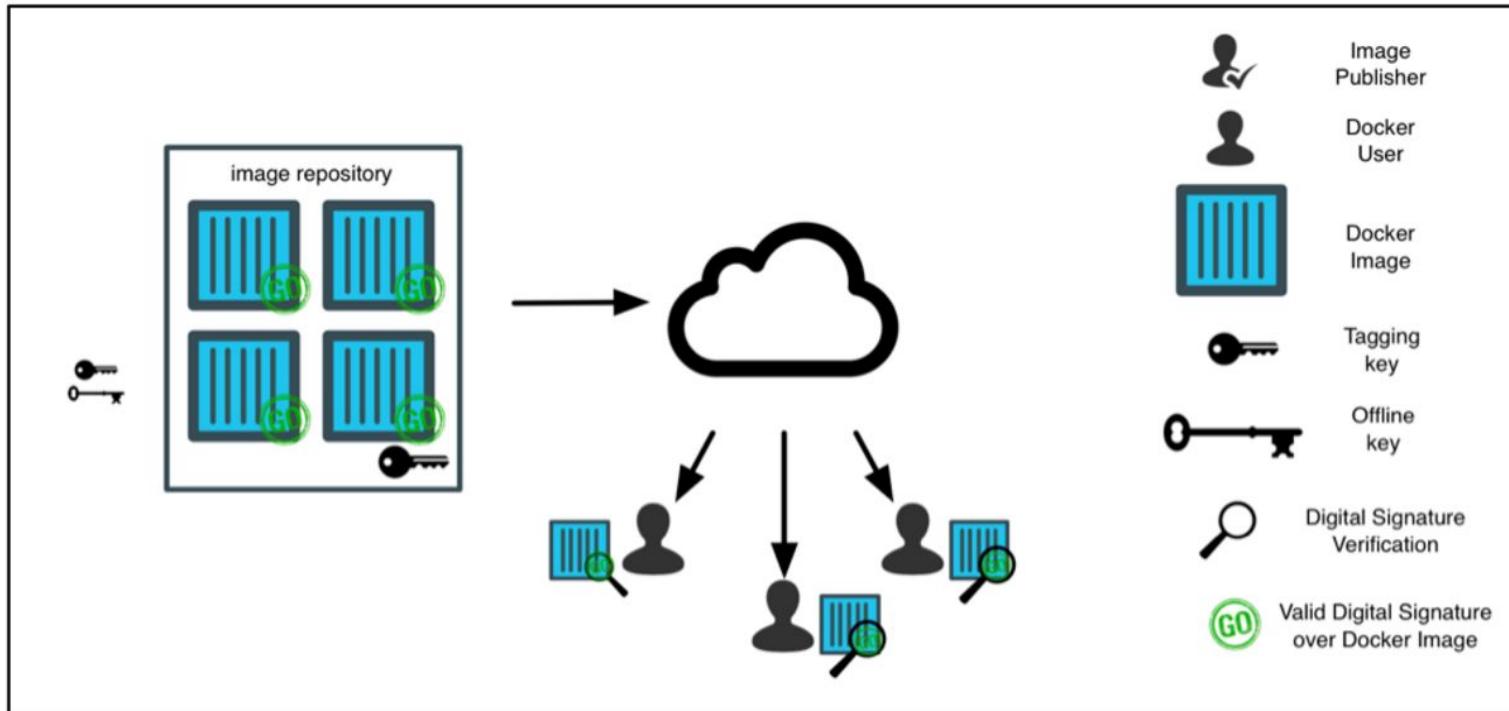


How it works for image consumers

- Once Docker Content Trust has been enabled, only signed images can be used
- If you try to pull an image or run a container from an image that is not signed, it will fail
- The first time you pull an image, trust is established to the repository with the offline key
- All subsequent interactions with that image require a valid signature verification from that same publisher
- Once trust is established, TUF will ensure integrity and freshness on the content, via the use of a timestamp key
- Docker uses and manages the timestamp key



How it works for image consumers



Running containers from signed images

If content trust has not been enabled on the shell, the following command will run a container using the signed image called myimage

```
docker run -it --disable-content-trust=false myimage
```

If content trust has been enabled and you wish to run a container from an unsigned image

```
docker run -it --disable-content-trust unsigned_image
```



Signed and unsigned tags

- The same image tag can be signed and unsigned
- Allows for iteration over the unsigned tag
- The signed tag could represent the completed version of that image
- Users with content trust enabled will only get the latest signed tag of that image



Managing your keys

- Do not forget the passphrase
- Back up your keys
- Keys are stored in `~/.docker/trust/private` folder. You can tar this directory into an archive
- Losing a key means that every consumer will get an error when trying to use images they have already downloaded
- To recover a lost key, contact Docker Support (support@docker.com)



Building images with content trust enabled

- When building an image from a Dockerfile, the same content trust rules apply
- The FROM instruction will only be allowed to pull signed images

In this example, if content trust is enabled, the 1.0 tag of the trainingteam/myjava image must be signed. Otherwise trying to build the image will result in failure

```
FROM trainingteam/myjava:1.0
RUN ...
CMD ...
```



Content Trust Exercise

https://github.com/docker/dceu_tutorials/blob/master/5-content-trust.md



Further reading

- For more about how Docker Content Trust enhances security check out <https://blog.docker.com/2015/08/content-trust-docker-1-8/>
- Contains examples of how signed images and the key system protect against various attacks

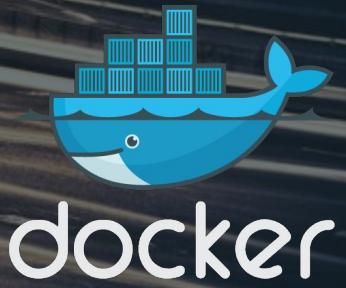


Module Summary

- Docker Content Trust allows us to ensure the integrity and publisher of Docker images through a signing process
- Publishers enable content trust to sign their images
- Consumers enable content trust to ensure that only signed images are used



Docker Trusted Registry



Module objectives

In this module we will:

- Learn about the new features of Docker Trusted Registry
- Outline the general installation and configuration requirements
- Demonstrate how to configure authentication on Docker Trusted Registry
- Create Organizations, Teams, and Repos.
- Push images into our Docker Trusted Registry registry
- Pull images from our Docker Trusted Registry registry



What is Docker Trusted Registry

Docker Trusted Registry (DTR) is a registry server that you can run securely on your own infrastructure

- What features does DTR include?
 - Image registry to store images
 - Organizations, Teams, Members
 - Pluggable storage drivers
 - API and Web Based GUI for Config
 - Easy and transparent upgrades
 - Built in system usage metrics dashboard
 - Logging



DTR Primary Usage Scenarios

CI/CD with Docker

- Centrally located base images
- Store individual build images
- Pull tested images to production
- Developer workflow is a commit

Containers as a Service

- Deploy Jenkins executors or Hadoop nodes
- Instant-on developer environment
- Select curated apps from a catalog
- Dynamic composition of micro-services from catalog (“PaaS”)



DTR < 1.4

General Features

- Admin & System Health UI
- Accounts & repos API (RBAC to the repo level)

CI/CD with Docker

- Registry Storage Status
- Registry v2 API and v2 Image Support

Registry as a Marketplace

- LDAP/AD and basic user Authentication
- Built-in RBAC for Admin, User (R/W), Pull-Only (R/O)

Platform Features

- Storage Drivers for Local file system, S3, and Azure
- One click install, one-click upgrade
- Scales to 300 concurrent pulls per instance
- Support Tooling
- Audit Logs for User actions and API access
- Support for Ubuntu, RHEL, CentOS



DTR 1.4

General Features

- Accounts & repos groups UI

CI/CD with Docker

- Image garbage collection
- Visual API runner & docs

Registry as a Marketplace

- Image deletion from index
- Search & browse index & UI

Platform Features

- Image Provenance
 - Docker Notary Support



Installation requirements

- DTR license
- Commercially supported version of Docker Engine (CS Engine)
- Supported OS (Ubuntu/RHEL/Centos..etc)

The screenshot shows a web browser window with the Docker Trusted Registry trial sign-up page. The header includes the Docker logo, 'Dashboard', 'Explore', 'Organizations', a search bar, a 'Create' button, and a user profile for 'nicolaka'. The main content area has a heading 'Register for a trial of Docker Trusted Registry'. Below it, there's a section for signing up for a free 30-day trial, mentioning access to tools for managing Docker images. It asks to attach a license to Docker Hub and provides fields for contact information: First Name, Last Name, Company, Company Email Address, Phone Number, and Country (set to United States). A large blue 'Start Your Free Trial' button is at the bottom. To the right, a sidebar titled 'What's Included' lists '1 Docker Trusted Registry', '10 Commercially supported Docker Engines with patches and hotfixes', and 'Support from the Docker community'. The Docker logo is also present in the sidebar.



Configuration overview

- Once DTR is installed and running you will need to configure
 - Domain and ports
 - License
 - SSL certificates
 - Image storage
 - Authentication method
- All done through the web based admin console
- Access the GUI by pointing to the server URL on your browser



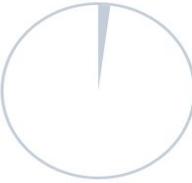
Web admin interface

TRUSTED REGISTRY Dashboard Repositories Organizations Settings Logs  gordon

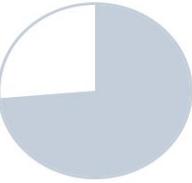
Dashboard

Host Status

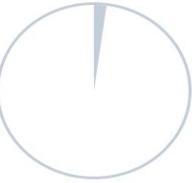
RAM
76.7MB/3.7GB



Storage
7.1GB/9.7GB



CPU
2.14/100%



Container Status

Admin Server		Auth Server	
			
RAM 18.4MB/3.7GB	Network 176.2KB/s	RAM 1.6MB/3.7GB	Network 341.9B/s
CPU 1.68%	CPU 0.01%		



Configuring SSL certificates

- SSL needs to be configured between
 - The DTR registry and all Docker Engines that want to connect to it
 - The DTR admin server and your web browser
- During installing, self signed certificates are auto generated
- Can also generate your own certificates and add them to DTR
 - Use your own private key infrastructure (PKI)
 - Use a Certificate Authority (CA)
- If using certificates from a trusted CA, you do not need to install them on each client Docker daemon
- For certificates from an untrusted CA, you need to install the certificate on each client Docker daemon by following the procedure at
<https://docs.docker.com/docker-hub-enterprise/configuration/#installing-registry-certificates-on-client-docker-daemons>



Adding your own certificate

- Done in the “Security” tab under “Settings”
- Must add the certificate and private key separately

The screenshot shows the Trusted Registry settings interface. At the top, there is a blue header bar with the word "Settings". Below it is a navigation bar with tabs: General, Security (which is highlighted in blue), Storage, License, Auth, Garbage collection, and Updates. The main content area has a light gray background. It contains the following text:

You can generate your own certificates for Trusted Registry using a public service or your enterprise's infrastructure.

SSL Certificate

The certificate that was issued by a Certificate Authority. If there are any intermediate certificates they should be included here in the correct order.

SSL certificate (hidden for security reasons)



Notary Integration (Experimental)

- Requires an on-prem Notary Server

Notary Server (experimental feature)

Notary server url. Note that for Notary signatures to show up in the Trusted Registry UI you must use the same domain name when pushing as the domain name configured in Trusted Registry. Ex. <https://172.17.42.1:4443>

Notary Verify TLS (experimental feature)

Whether or not to verify that the TLS certificate is valid for the Notary server. This is necessary for production environments.

Notary TLS Root CA (experimental feature)

The TLS certificate of the Certificate Authority used to verify Notary's certificate (if not already in operating system's CA store).

TLS certificate



Storage Backends

TRUSTED REGISTRY | Dashboard | Repositories | Organizations | Settings | Logs | gordon

Settings

General Security **Storage** License Auth Garbage collection Updates

Configure your storage to use the local filesystem, S3, or Azure

Fill out settings via a form

Storage Backend
Choose your storage backend

S3

S3 settings

AWS region
The name of the AWS region in which you would like to store objects
(for example us-east-1)

Bucket name
The name of your S3 bucket where you wish to store objects (needs to already be created prior to driver initialization)

Access Key
Your AWS access key



Garbage Collection and Image Deletion

- Soft Image Delete = Remove from UI but still on disk
- Hard Image Delete = Remove unused layers via garbage collection

The screenshot shows the Trusted Registry interface. At the top, there's a navigation bar with links for Dashboard, Repositories, Organizations, Settings, and Logs. A user icon labeled 'gordon' is also present. Below the navigation bar is a blue header bar with the word 'Settings'. Underneath this, there's a grey navigation bar with tabs: General, Security, Storage, License, Auth, Garbage collection (which is underlined in blue), and Updates. The main content area has a heading 'Trusted Registry can run garbage collection on your storage backend to remove deleted tags and images.' Below this, there's a section titled 'Schedule' with a note about cron expressions. To the right of the schedule note is a text input field containing '0 * * * *' and a 'Save' button.

Trusted Registry can run garbage collection on your storage backend to remove deleted tags and images.

Schedule

Cron schedule for garbage collection, such as '0 * * * *' to run daily at midnight. A cron expression represents a set of times in the series: hours, day of month, month, day of week.

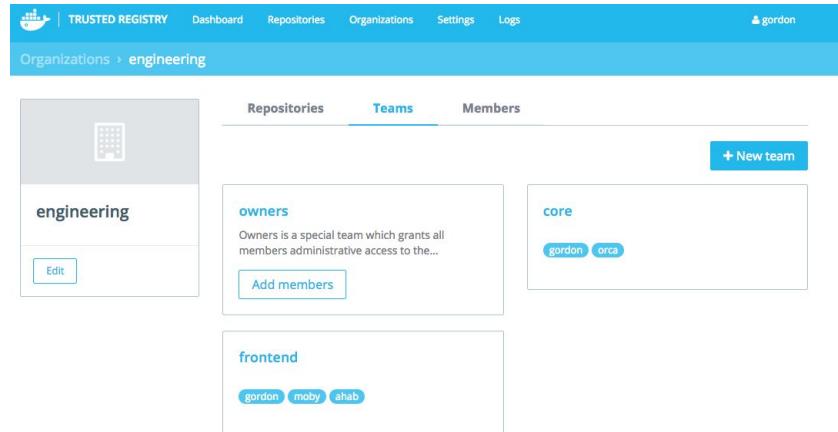
0 * * * *

Save



Organizations, Team, and Members

- Security, Isolation, Auditing
- Privileged members can create repos
- Team Collaboration
- LDAP Integration



The screenshot shows the Docker Trusted Registry interface. At the top, there's a navigation bar with links for TRUSTED REGISTRY, Dashboard, Repositories, Organizations, Settings, and Logs. A user icon for 'gordon' is also present. Below the navigation, the path 'Organizations > engineering' is shown. The main area has three tabs: 'Repositories', 'Teams', and 'Members'. The 'Teams' tab is selected. It displays three teams under the 'engineering' organization: 'owners', 'core', and 'frontend'. The 'owners' team is described as a special team granting administrative access. The 'core' team has members 'gordon' and 'orca'. The 'frontend' team has members 'gordon', 'moby', and 'ahab'. A blue button '+ New team' is located at the top right of the team list.



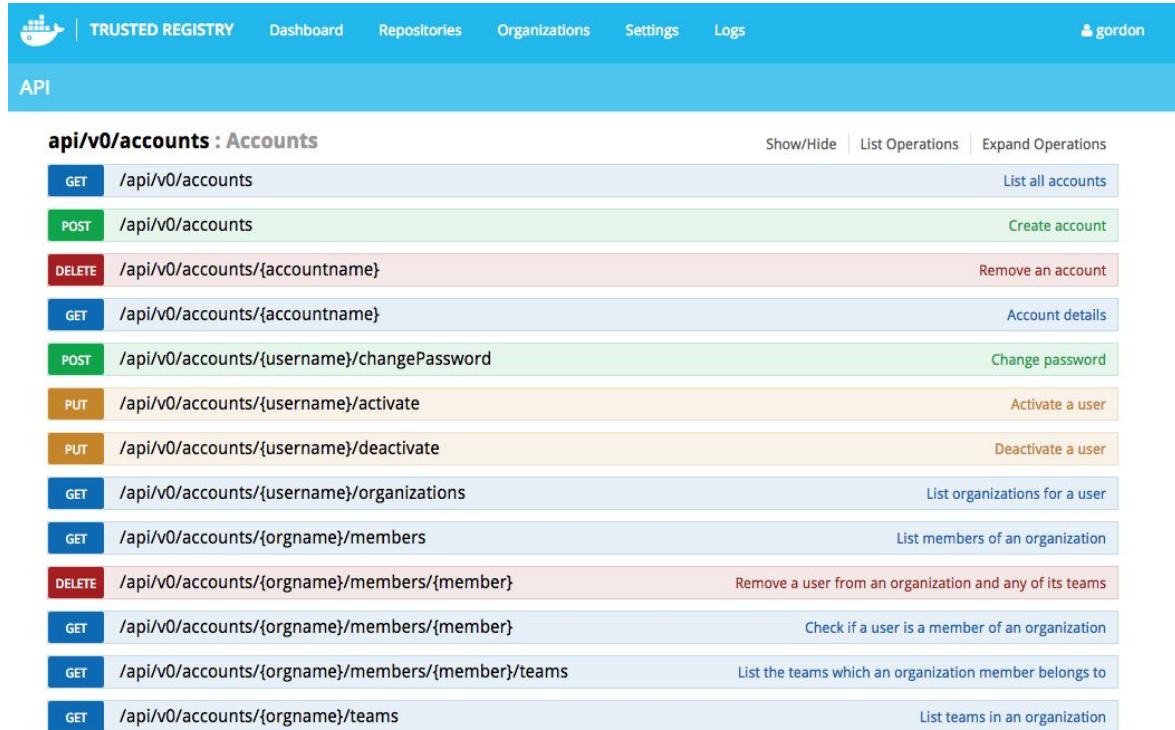
Repositories

- Public and Private
- Search in UI
- Team-level permissions
- Personal and Organization Repos
- UI Search

The screenshot shows a web interface for a Docker Trusted Registry. At the top, there's a navigation bar with links for TRUSTED REGISTRY, Dashboard, Repositories, Organizations, Settings, and Logs. On the far right, it shows a user icon and the name "gordon". Below the navigation bar, the word "Repositories" is centered in a blue header bar. Underneath, there's a search bar with tabs for "All", "My repositories", and "Shared with me", along with a magnifying glass icon. To the right of the search bar is a blue button with a plus sign and the text "New repository". The main content area lists several repository names, each followed by a small orange box containing the word "private":

- gordon/myrepo
- engineering/nginx
- engineering/wordpress-frontend
- gordon/ubuntu
- engineering/madrid
- engineering/barca

API Console + Docs



The screenshot shows the API Console interface for a 'TRUSTED REGISTRY'. The top navigation bar includes links for Dashboard, Repositories, Organizations, Settings, Logs, and a user profile for 'gordon'. The main header is 'API'.

api/v0/accounts : Accounts

Show/Hide | List Operations | Expand Operations

Method	Endpoint	Description
GET	/api/v0/accounts	List all accounts
POST	/api/v0/accounts	Create account
DELETE	/api/v0/accounts/{accountname}	Remove an account
GET	/api/v0/accounts/{accountname}	Account details
POST	/api/v0/accounts/{username}/changePassword	Change password
PUT	/api/v0/accounts/{username}/activate	Activate a user
PUT	/api/v0/accounts/{username}/deactivate	Deactivate a user
GET	/api/v0/accounts/{username}/organizations	List organizations for a user
GET	/api/v0/accounts/{orgname}/members	List members of an organization
DELETE	/api/v0/accounts/{orgname}/members/{member}	Remove a user from an organization and any of its teams
GET	/api/v0/accounts/{orgname}/members/{member}	Check if a user is a member of an organization
GET	/api/v0/accounts/{orgname}/members/{member}/teams	List the teams which an organization member belongs to
GET	/api/v0/accounts/{orgname}/teams	List teams in an organization



Logging Server

See below for streaming output from each of DTR's containers:

[Filter by container](#) [Filter by date](#)

-

2015-11-05T23:29:59.254402185Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created an account index! account=Gordon</code>
2015-11-05T23:29:59.458395724Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created an account index! account=Orca</code>
2015-11-05T23:29:59.622141930Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created an account index! account=Moby</code>
2015-11-05T23:39:17.071745756Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created an account index! account=engineering</code>
2015-11-05T23:43:25.711357456Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created an account index! account=sales</code>
2015-11-06T00:08:11.955800953Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created a repo index! repo="engineering/barca"</code>
2015-11-06T00:11:15.311537870Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created an account index! account="rm_me"</code>
2015-11-06T00:17:29.167311111Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created an account index! account=Ahab</code>
2015-11-06T00:20:02.234288105Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created a repo index! repo="sales/madrid"</code>
2015-11-06T00:34:02.339079824Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created a repo index! repo="engineering/corerepo"</code>
2015-11-06T00:35:28.043295618Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created a repo index! repo="engineering/admin_repo"</code>
2015-11-06T00:36:24.288163015Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created a repo index! repo="engineering/admin_repo_public"</code>
2015-11-06T00:36:48.909279619Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created a repo index! repo="engineering/org_level_repo"</code>
2015-11-06T00:39:47.222720767Z	<code>text=INFO [1.4.0-rc3-003149_ga20ae3e] created a repo index! repo="engineering/private_repo"</code>



Setting Up DTR

Tasks 1-4 in https://github.com/docker/dceu_tutorials/blob/master/3-dtr.md

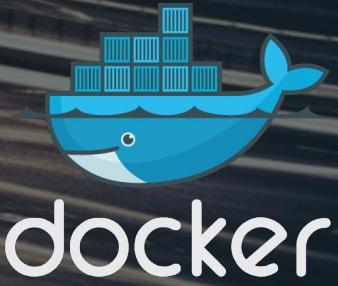


Module summary

- DTR is *registry server that you can run securely on your own infrastructure*
- *DTR is commercially supported*
- DTR requires a commercially supported version of the Docker Engine running on either Ubuntu 14.04 LTS or RHEL 7.0, 7.1
- The DTR server is made up of several components each running in their own container



Docker Swarm



Module objectives

In this module we will:

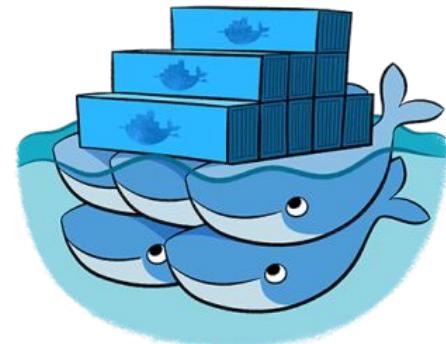
- Learn how to install Docker Swarm
- Setup a Swarm cluster using the hosted discovery backend
- Explain and try the different scheduling strategies when running containers in a Swarm cluster
- Explain and try the different filtering options when running containers in a Swarm cluster



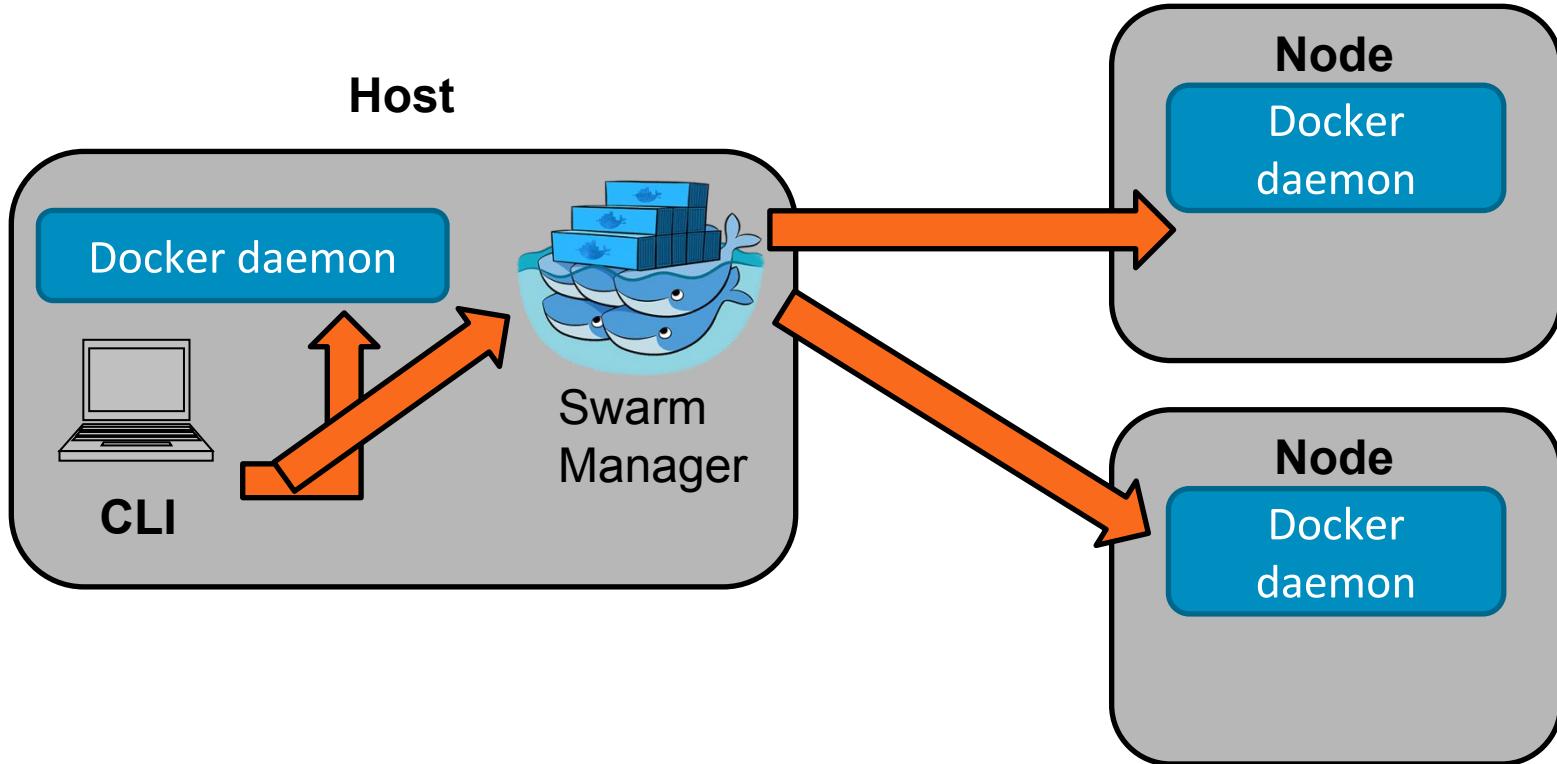
What is Docker Swarm?

Docker Swarm is a native tool that clusters Docker hosts and schedules containers on them

- Turns a pool of Docker host machines into a single virtual host
- Allows us to distribute container workloads across multiple machines running in a cluster
- Serves the standard Docker API
- Ships with simple scheduling and discovery backend



How Swarm works



Discovery backends

- Supports many discovery backends
 - Hosted discovery
 - etcd
 - Consul
 - ZooKeeper
 - Static files



Installing Swarm

- There are two ways to install Swarm
 - Install the Swarm binary
 - Run Swarm in a container using the Swarm image
- The Swarm binary needs to be installed on every machine that is going to be part of the cluster.
- For instructions on installing the binary see <https://github.com/docker/swarm>
- Using the Swarm image is a convenient option as we don't have to install all the prerequisites for running the Swarm binary



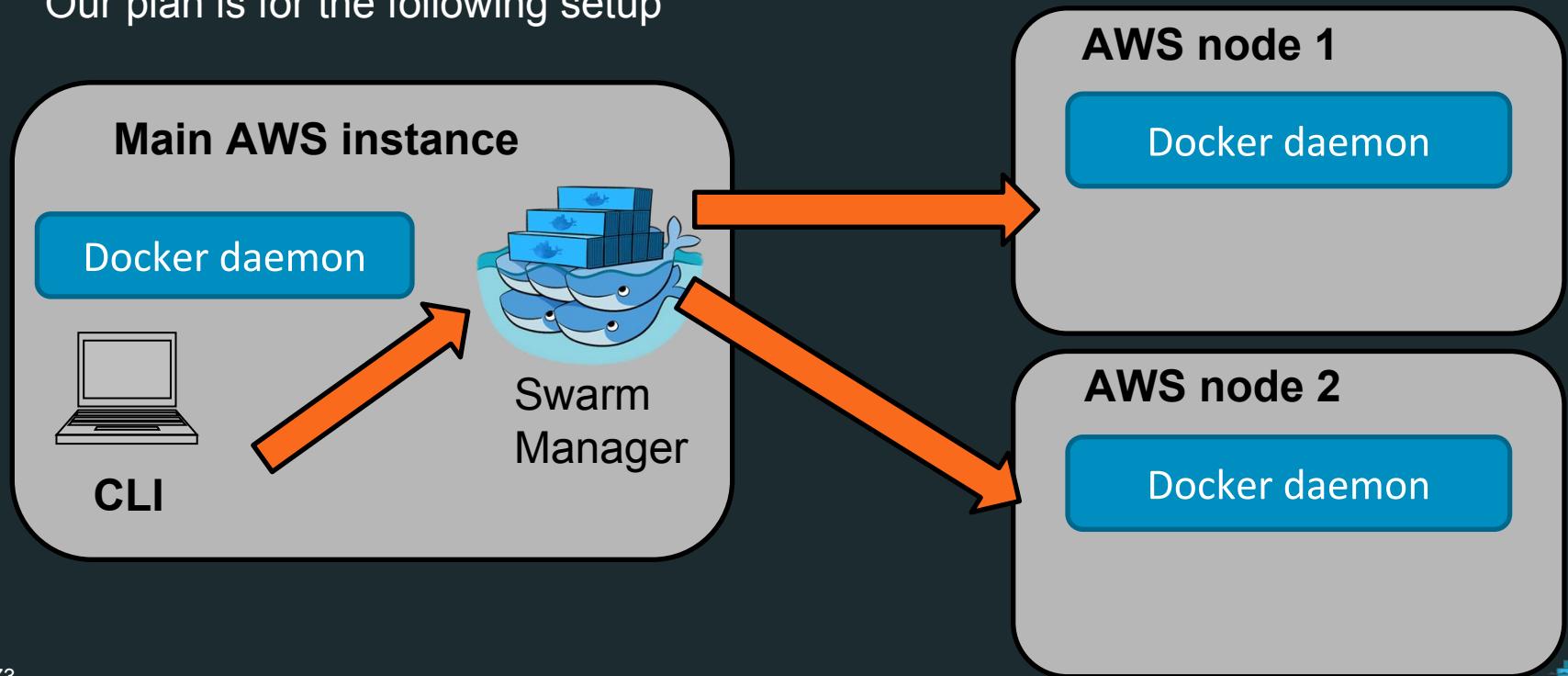
Installing swarm from image

- Most convenient option is to use the Swarm image on Docker Hub
<https://registry.hub.docker.com/u/library/swarm/>
- Swarm container is a convenient packaging mechanism for the Swarm binary
- Swarm containers can be run from the image to do the following
 - Create a cluster
 - Start the Swarm manager
 - Join nodes to the cluster
 - List nodes on a cluster



Using the hosted discovery backend

- In this section, we will setup a Swarm cluster using the hosted discovery backend
- Our plan is for the following setup



Setup the cluster

- On the machine that you will use as the Swarm master, run a command to create the cluster
- Start Swarm master
- For each node with Docker installed, run a command to start the Swarm agent
- **Note:** Agents can be started before or after the master



Create the Swarm cluster

- `swarm create` command will output the cluster token
- Token is an alphanumeric sequence of characters that identifies the cluster when using the hosted discovery protocol
- Copy this number somewhere

Run a container using the `swarm` image. We run the `create` command of the Swarm application inside and get the output on our terminal

`--rm` means to remove the container once it has finished running

```
docker run --rm swarm create
```



Start the Swarm manager

- Run a container that runs the `swarm` manager
- Make sure to map the `swarm` port in the container to a port on the host
- Command syntax is
`swarm manage token://<cluster token>`

Running the `swarm manage` command via a swarm container

```
docker run -d -P swarm manage token://<cluster token>
```



Connect a node to the cluster

- Run a container that runs the `swarm join` command
- Specify the IP address of the node and the port the Docker daemon is listening on
- **Note:** Your Docker daemon on the machine must be configured to listen on a TCP port instead of just on the unix socket

```
docker run -d swarm join  
  --addr=<node ip>:<daemon port> \ token://<cluster  
token>
```



List nodes in the cluster

- The `swarm list` command will output a list of the IP addresses of all the running nodes
- Syntax

```
swarm list token://<cluster token>
```

Running the `swarm list` command via the `swarm` container

```
docker run --rm swarm list token://<cluster_id>
```



Connect the Docker client to Swarm

- Point your Docker client to the Swarm manager container
- Two methods:
 - Configuring the `DOCKER_HOST` variable with the Swarm IP and port
 - Run `docker` with `-H` and specify the Swarm IP and port
- Look at the container port mapping to find the Swarm port

Configure the `DOCKER_HOST` variable

```
export DOCKER_HOST=127.0.0.1:<swarm port>
```

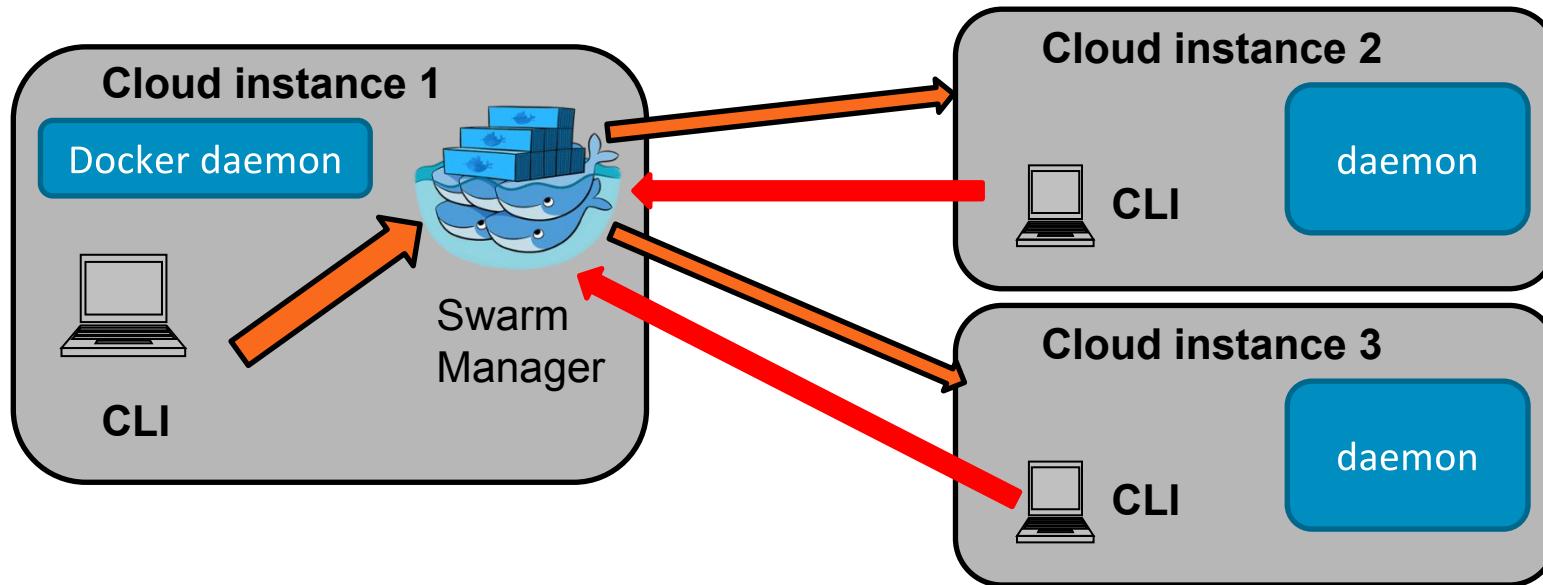
Run `docker` client and specify the daemon to connect to

```
docker -H tcp://127.0.0.1:<swarm port>
```



Connect the Docker client to Swarm

- The Docker client on each node can also be connected to the Swarm manager



Verify the Docker client

- To ensure your Docker client is connected to Swarm, run `docker version`
- Server version should indicate Swarm

```
student@master:~$ docker version
Client:
  Version:      1.9.0
  API version:  1.21
  Go version:   go1.4.2
  Git commit:   76d6bc9
  Built:        Tue Nov  3 17:48:04 UTC 2015
  OS/Arch:      linux/amd64

Server:
  Version:      swarm/1.0.0
  API version:  1.21
  Go version:   go1.5.1
  Git commit:   087e245
  Built:
  OS/Arch:      linux/amd64
```



Checking your connected nodes

- Run `docker info`
- Since client is connected to Swarm, it will show the nodes

```
student@DockerTraining:~$ docker info
Containers: 4
Images: 3
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 2
  node1: 104.236.179.194:2375
    └ Containers: 2
      └ Reserved CPUs: 0 / 1
      └ Reserved Memory: 0 B / 514.5 MiB
      └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-58-generic, operatingsystem=Ubuntu 14.04.3 LTS, storagedriver=aufs
  node2: 104.236.142.73:2375
    └ Containers: 2
      └ Reserved CPUs: 0 / 1
      └ Reserved Memory: 0 B / 514.5 MiB
      └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-58-generic, operatingsystem=Ubuntu 14.04.3 LTS, storagedriver=aufs
CPUs: 2
Total Memory: 1.005 GiB
Name: 9af958b7e141
```



Run a container in the cluster

- Can use the docker run command
- Swarm master decides which node to run the container on based on your scheduling strategy
- Running docker ps will show which node a container is on

CONTAINER ID	IMAGE	COMMAND	...	PORTS	NAMES
a46d77a60121	nginx:latest	"nginx -g 'daemon off';	...	80/tcp, 443/tcp	node1/goofy_morse
6387c6790ce8	nginx:latest	"nginx -g 'daemon off';	...	80/tcp, 443/tcp	node2/adoring_albattani
53f762457a46	tomcat:latest	"catalina.sh run"	...	104.236.162.207:32768->8080/tcp	node1/grave_elion



Stop and start containers in the cluster

- Use the docker stop and docker start commands
- If you specify a container using the container name, you do not need to specify the node in the name
- A stopped container will start on the node it was previously running on

CONTAINER ID	IMAGE	COMMAND	... PORTS	NAMES
a46d77a60121	nginx:latest	"nginx -g 'daemon off'; ..."	80/tcp, 443/tcp	node1/goofy_morse
6387c6790ce8	nginx:latest	"nginx -g 'daemon off'; ..."	80/tcp, 443/tcp	node2/adoring_albattani
53f762457a46	tomcat:latest	"catalina.sh run"	104.236.162.207:32768->8080/tcp	node1/grave_elion

Stop the container named grave_elion

```
docker stop grave_elion
```



Running Docker commands with Swarm

- You can run all the regular Docker commands when the Docker client is connected to the Swarm manager
- For example, to check the container log, you still use
`docker logs <container name>`
- To inspect container details
`docker inspect <container name>`
- If you run `docker images`, you will get a combined list of images from every node



Stop and start containers in the node

- If you login to the individual node, you can still control the Docker daemon there on its own.
- If you run a new container directly on the node, Swarm will detect it and still make it part of the cluster
- If you stop a container on the node, the container state will be reflected in the Swarm manager (i.e if you run docker ps against the Swarm manager, the container will be listed as stopped)



Scheduling containers in the cluster

- In your previous exercise, you would have noticed that the two containers you ran, were spread across both nodes
- In this section, we will learn how the Swarm manager decides which node your container runs on



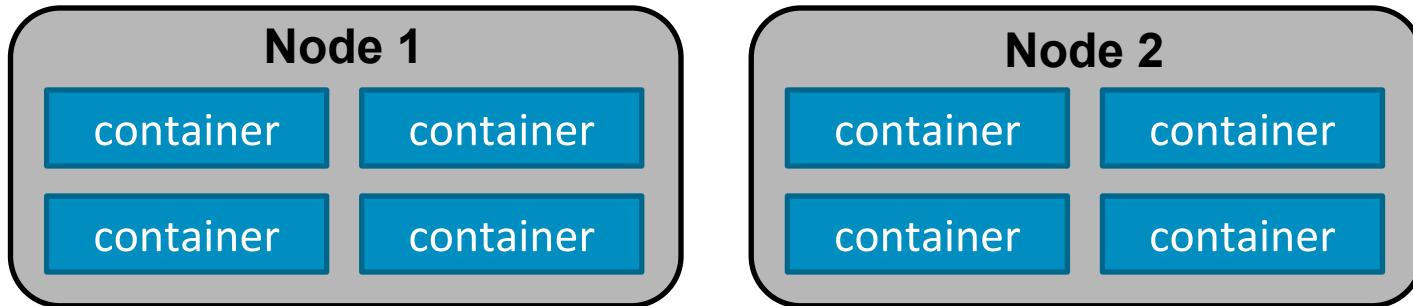
Overview of scheduling strategies

- The Docker Swarm scheduler ranks nodes based on a number of different strategies / algorithms
- When you run a container, Swarm will place it in the node with the highest rank
- How the rank is calculated depends on your selected strategy
- Strategies are
 - Spread (default strategy)
 - Binpack
 - Random



Spread Strategy

- The spread strategy ranks nodes based on the number of containers
- Swarm will schedule the container on the node that has the least number of containers running
- If multiple nodes have the same least number of containers running, Swarm will pick one of those nodes at random



Binpack strategy

- When using the `binpack` strategy, Swarm will try to fit as many containers into a node as possible, before using another node
- Optimizes for the container that is the most packed
- Swarm will continue to schedule containers on the same node until there are insufficient resources (CPU, RAM) on that node
- **Note:** when using `binpack` you have to specify the memory and/or CPU allocation when running containers. Otherwise Swarm will pack every container into the one host regardless of its resources.



Specifying memory requirements

- When running a container we can specify the memory limit to be allocated by using the `-m` option and specify the memory in either bytes, kilobytes, megabytes or gigabytes
- Swarm will only schedule the container in a host that meets the memory limit
- For example, if we have 2 hosts with 2 GB of RAM each and we specify to run a container with 3 GB of RAM, Swarm will not schedule the container as no host can meet the requirement

Run a container with a memory limit of 512mb

```
docker run -d -m 512MB nginx
```

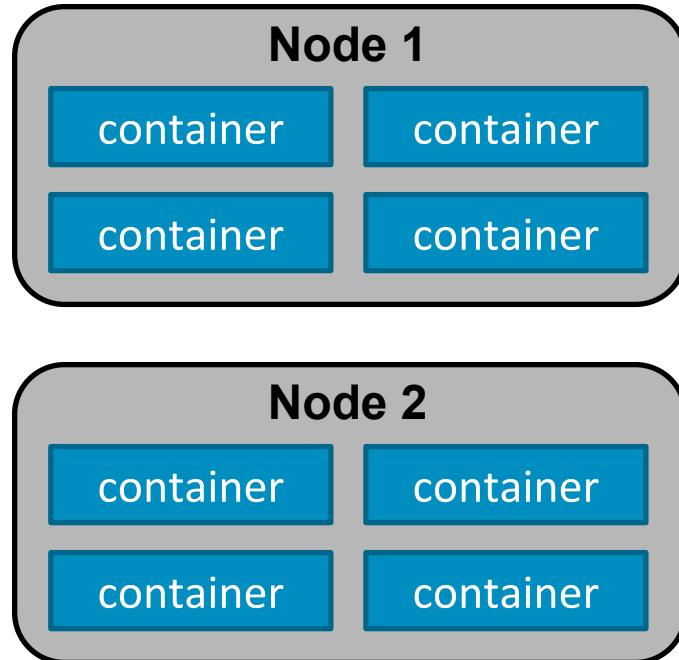
Run a container with a memory limit of 2GB

```
docker run -d -m 2GB tomcat
```



Binpack example

- Both nodes have 2GB of RAM
- We will run containers with a memory limit of 512 MB
`docker run -m 512MB ...`
- First container will run on a randomly chosen node
- Next containers will run on that same node until the resource constraints can no longer be met



Specifying your strategy

- To specify your scheduling strategy, use the `--strategy` option when running the `swarm manage` command
- If strategy is not specified, Swarm defaults to spread

```
docker run -d -P swarm manage token://<token>
  --strategy binpack
```

```
docker run -d -P swarm manage token://<token>
  --strategy spread
```

```
docker run -d -P swarm manage token://<token>
  --strategy random
```



Checking nodes and resources

- docker info shows
 - how many containers and in the cluster
 - Reserved RAM
 - Reserved CPUs
- Also shows the configured scheduling strategy

```
johnnytu@docker-ubuntu:~$ docker info
Containers: 0
Strategy: binpack
Filters: affinity, health, constraint, port, dependency
Nodes: 3
node1: 104.236.162.207:2375
  L Containers: 0
  L Reserved CPUs: 0 / 1
  L Reserved Memory: 0 B / 514.5 MiB
node2: 104.236.163.107:2375
  L Containers: 0
  L Reserved CPUs: 0 / 1
  L Reserved Memory: 0 B / 514.5 MiB
node3: 104.131.142.17:2375
  L Containers: 0
  L Reserved CPUs: 0 / 1
  L Reserved Memory: 0 B / 1.019 GiB
```



Binpacking containers

- Output of docker info after running two NGINX containers with 200 mb memory limit

```
docker run -d -m 200MB nginx
```

```
Nodes: 3
node1: 104.236.162.207:2375
  ↳ Containers: 0
  ↳ Reserved CPUs: 0 / 1
  ↳ Reserved Memory: 0 B / 514.5 MiB
node2: 104.236.163.107:2375
  ↳ Containers: 2
  ↳ Reserved CPUs: 0 / 1
  ↳ Reserved Memory: 400 MiB / 514.5 MiB
node3: 104.131.142.17:2375
  ↳ Containers: 0
  ↳ Reserved CPUs: 0 / 1
  ↳ Reserved Memory: 0 B / 1.019 GiB
```



Binpacking containers (cont'd)

- The next NGINX container we run with a 200 mb limit will go to node 1, because node 2 can no longer accommodate the resource requirement.

```
Nodes: 3
node1: 104.236.162.207:2375
  L Containers: 1
  L Reserved CPUs: 0 / 1
  L Reserved Memory: 200 MiB / 514.5 MiB
node2: 104.236.163.107:2375
  L Containers: 2
  L Reserved CPUs: 0 / 1
  L Reserved Memory: 400 MiB / 514.5 MiB
node3: 104.131.142.17:2375
  L Containers: 0
  L Reserved CPUs: 0 / 1
  L Reserved Memory: 0 B / 1.019 GiB
```



Which strategy to use?

- Spread strategy is good if you want to distribute containers across your resources.
- If one node goes down, you won't lose that many containers
- To take advantage of this, you will need a lot of nodes
- The binpack strategy makes more use of each node and saves unused machines for containers with higher requirements
- Binpack strategy does not require as much nodes but if you lose a node, you will lose more containers



The random strategy

- Does not use any ranking algorithm
- Simply schedules a container onto a random node in the cluster
- Mainly intended for debugging purposes



Filters

- In the next section we will look at using filters to control how Swarm schedules the containers we run
- We will cover three types of filters
 - Constraint
 - Affinity
 - Port



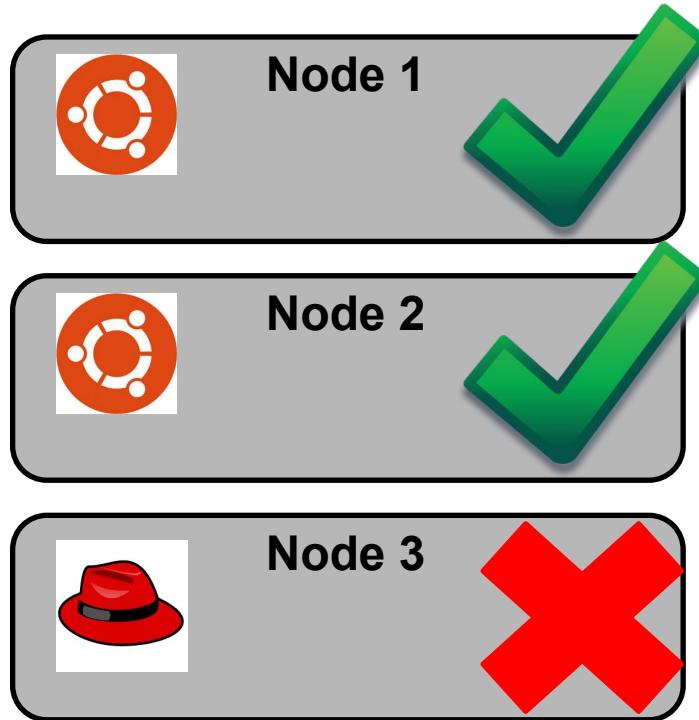
Purpose of filters

- Filters are used to determine a subset of nodes for which Swarm can schedule containers on
- The Swarm scheduler applies the configured filter, before it actually applies the scheduling strategy
- The filter will eliminate the nodes that do not match the criteria, then the scheduling strategy will select an appropriate node from the remaining options



Example of filtering

- Three nodes to schedule containers on
- Node 1 and 2 are Ubuntu hosts
- Node 3 is RHEL
- Apply a filter, saying operating system must be Ubuntu
- Node 3 fails and is sidelined
- Node 1 and 2 will be used



Constraint filter

- A **constraint filter** will filter nodes based on labels that have been applied to the Docker daemon
- A **label** is a key value pair that defines a characteristic of the node the daemon is running on
- Labels can be used to indicate any characteristic of the host
 - CPU chipset (indicate a host with more powerful CPU)
 - Storage type (indicate a host using SSD's or disk drive)
 - Host region (Is our server is US or EU or APAC etc...)
 - Environment (QA, staging, production etc...)



Labelling a node

- Run the Docker daemon with the `--label` option and then specify the key value pair
- If your node is already part of a cluster, you have to restart the Swarm manager in order for it to pick the new label
- To check labels on the Daemon, point the Docker client at the Daemon and run `docker info`

Run the Docker daemon with a label to indicate that the host is in the US region

```
docker daemon --label region=us
```

Example with multiple labels. The second label indicates that the host uses SSD's for storage

```
docker daemon --label region=us --label storage=ssd
```



Specifying a constraint filter

- Filters (constraint and affinity) are passed as environment variables to containers
- On the `docker run` command, use the `-e` option and then specify:
`constraint:<key><operator><value>`
- Valid operators are `==` and `!=`
- If multiple constraints are specified the node must satisfy all

Run an NGINX container on a node where the Docker daemon has label of storage = ssd

```
docker run -d -e constraint:storage==ssd nginx
```

Example with multiple constraints

```
docker run -d -e constraint:storage==ssd constraint:region==us nginx
```



Standard constraints

- Standard constraints are built in tags we can use without having to define any labels
- The constraints are sourced from the output of docker info
 - storagedriver
 - executiondriver
 - kernelversion
 - operatingsystem

```
johnnytu@docker-ubuntu:~$ docker info
Containers: 4
Images: 142
Storage Driver: aufs
  Root Dir: /var/lib/docker/aufs
  Backing Filesystem: extfs
  Dirs: 150
  Dirperm1 Supported: false
Execution Driver: native-0.2
Kernel Version: 3.13.0-43-generic
Operating System: Ubuntu 14.04.1 LTS
CPUs: 1
Total Memory: 490 MiB
Name: docker-ubuntu
```



Specifying a node constraint

- We can also use the constraint filter to directly specify which node to run or not to run the container on
- Expression is
 - e constraint:node<operator><node>

Run container on node 1

```
docker run -d -e constraint:node==node1 nginx
```

Run container on any node other than node 1

```
docker run -d -e constraint:node!=node1 nginx
```



Affinity filter

- An affinity filter allows Swarm to schedule a container on a node which already has a specified container or image
- For example:
 - Run a container on a node which has a MySQL database container
 - Run a container on a node which already has the Tomcat image
- To specify an affinity filter, use the `-e` option
`-e affinity:<key><operator><value>`



Container affinity

- To schedule a container next to another container use:
affinity:container==<container name or id>

Run a container in a node where there is a container called dbms

```
docker run -d -e affinity:container==dbms tomcat
```



Image affinity

- To schedule a container in a node that has a specified image we use:
affinity:image==<image name or id>

Run a Tomcat container in any node which already has the Tomcat image pulled.

```
docker run -d -e affinity:image==tomcat tomcat
```

Example with a specific image tag

```
docker run -d -e affinity:image==tomcat:6 tomcat:6
```



Affinity and constraint regex

- Constraint and affinity filters values can be specified with a regular expression or globbing pattern (wildcards).
- The regex is of the form /regex/
- Uses Go's regular expression syntax

Run a tomcat container on any node with either the Tomcat 6 or Tomcat 7 image

```
docker run -d -e affinity:image==/tomcat:[67] / tomcat
```

Run container on any node with a region label value that is prefixed with “us”

```
docker run -d -d constraint:region==us* nginx
```



Soft vs hard affinities and constraints

- **Hard** affinity or constraint
 - if no node can meet the filter, Swarm will not schedule the container at all
- **Soft** affinity or constraint
 - If no node meets the filter, Swarm ignores the filter and just uses its configured scheduling strategy
- Default is hard affinity
- To use soft affinity put the “~” symbol after the operator
- Soft constraint example:
`docker run -d -e constraint:region==~apac tomcat`



Port filtering

- Applying a port filter means that swarm will only consider nodes, where the specified port is available on the host
- Port filtering is automatically applied if a manual port mapping is specified when running a container

**Here we map port 80 on the NGINX container to port 80 on the host.
Swarm will only select a node which has port 80 available**

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



Dependency filter

- Containers that have a dependency to another container will be scheduled on the same node as the dependency
- Container dependencies are based on
 - Shared volumes (`--volumes-from=<container name>`)
 - Links (`--link=<container name>:<alias>`)

Run a Tomcat container called appserver

```
docker run -d -name appserver tomcat
```

Run an NGINX server and link it to our “appserver”. Because we specify a link, Swarm will schedule it on the same node as our appserver

```
docker run -d --link=appserver:tomcat nginx
```



Dependency filter (cont'd)

- If a container cannot be scheduled on the same node as the dependency, the container will not be created
- If there is more than one dependency, all dependency containers must be scheduled on the same node

Swarm will schedule the container on the same node as both the linked containers. If the mysql and redis containers are on different nodes, Swarm will not schedule the container

```
docker run -d --link=mysql:mysql --link=redis:redis \
    nginx
```



Further notes on Swarm

- In this section
 - Outline how Swarm works with Docker Networking
 - References to other discovery backends
 - Setting up TLS for Swarm
 - Using Docker Machine to create a Swarm cluster



Swarm and Networking

- Swarm is compatible with the new networking model in Docker 1.9
- By default, Swarm will create multi-host networks using the overlay driver
 - You must ensure that the Docker daemon on each node is connected to a key-value store (as explained in Module 9: Container Networking)

Running docker network ls when connected to Swarm, will list all networks in all Swarm nodes

```
student@masterhost:~$ docker network ls
NETWORK ID      NAME      DRIVER
b510a385302d    node1/bridge    bridge
98041ae809b3    node2/none     null
f3d7dde7c511    node2/host     host
787f6fe62a35    node2/bridge    bridge
49958db81312    node1/none     null
a1135d73efb7    node1/host     host
```



Creating a multi-host network from Swarm

- Use the same docker network create command and specify the network name
- Overlay driver is used by default

```
student@masterhost:~$ docker network create swarm_multi  
2a0797f00bacb6308f733f04cb85983e91d4ba48e09760fb08abe445856e6ef  
student@masterhost:~$ docker network ls  
NETWORK ID          NAME            DRIVER  
a1135d73efb7        node1/host      host  
98041ae809b3        node2/none      null  
f3d7dde7c511        node2/host      host  
787f6fe62a35        node2/bridge    bridge  
2a0797f00bac        swarm_multi    overlay  
b510a385302d        node1/bridge    bridge  
49958db81312        node1/none      null
```

Our new multi-host network 



Running containers on multi-host network

- Specify your multi-host network with `--net=<network name>`
- Remember to give your container a custom name
- The Swarm scheduler will decide which node the container runs on
- Containers on different nodes can communicate with each without the need for port mapping
- In the following example, we start an NGINX container and Swarm schedules it on Node 1

```
student@masterhost:~$ docker run -d --name nginx --net swarm_multi nginx  
536f7a4708ea1443e067e5c549e8143191bb912752b60f0f8feb0c996f9ba96a
```

```
student@masterhost:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	PORTS	NAMES
536f7a4708ea	nginx	"nginx -g 'daemon off'	80/tcp, 443/tcp	node1/nginx



Running containers on multi-host network

Let's inspect our nginx container

```
student@masterhost:~$ docker inspect nginx
[
{
...
{
  "Networks": {
    "swarm_multi": {
      "EndpointID": "9914ad736bfb38c41fc0678345c8421cc14ca55e9d1e8eafef09e890d8d4f890",
      "Gateway": "",
      "IPAddress": "10.0.0.2",
      "IPPrefixLen": 24,
      "IPv6Gateway": "",
      "GlobalIPv6Address": "",
      "GlobalIPv6PrefixLen": 0,
      "MacAddress": "02:42:0a:00:00:02"
    }
  }
}
```



Cross node container communication

- Let's run another container. This time, we will use Ubuntu and specify the same multi-host network
- You should notice the second container being scheduled into Node 2

```
student@masterhost:~$ docker run -itd --name client --net swarm_multi ubuntu:14.04
1a495a2f31d25e10cda940e3c1235316cfde04bf6b61a17f7403da475b668d27
student@masterhost:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             PORTS               NAMES
1a495a2f31d2        ubuntu:14.04       "/bin/bash"         80/tcp, 443/tcp   node2/client
536f7a4708ea        nginx              "nginx -g 'daemon off'"   node1/nginx
```



Cross node container communication (cont'd)

- We should see an entry for our nginx container in the /etc/hosts file of our Ubuntu container

```
student@masterhost:~$ docker exec -it client bash
root@1a495a2f31d2:#
root@1a495a2f31d2:~# cat /etc/hosts
10.0.0.3      1a495a2f31d2
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
10.0.0.2      nginx.swarm_multi
10.0.0.2      nginx
```



Cross node container communication (cont'd)

```
root@1a495a2f31d2:/# curl nginx
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
...
</html>
root@1a495a2f31d2:/#
```



Creating a bridge network with Swarm

- You can create a bridge network for a specific node in your cluster by specifying the bridge driver and the node name
- For example

```
docker network create -d bridge node1/mybridge
```
- If the node name is not specified, Swarm will create the network on a random node



Creating a bridge network with Swarm

```
student@masterhost:~$ docker network create -d bridge node1/my-app-net  
261b9c2e19811b9c7570358a88f0c199557bbfa481cac206ea56c33793c988a0
```

```
student@masterhost:~$ docker network ls
```

NETWORK ID	NAME	DRIVER
6dc031217adc	node2/docker_gwbridge	bridge
b510a385302d	node1/bridge	bridge
f3d7dde7c511	node2/host	host
2a0797f00bac	swarm_multi	overlay
49958db81312	node1/none	null
a1135d73efb7	node1/host	host
98041ae809b3	node2/none	null
787f6fe62a35	node2/bridge	bridge
6bb265220b9c	node1/docker_gwbridge	bridge
261b9c2e1981	node1/my-app-net	bridge



Running container on a bridge network

- To run a container on your user defined bridge network you need to specify:
 - A node constraint to indicate which node Swarm should schedule on
 - The name of the bridge network
- If you don't specify a node constraint, Swarm might try to schedule on the container on a node where the specified bridge network is not present

Example of specifying a user defined network along with the node constraint. Note that you don't need to indicate the node on the network name, unlike when we created the network

```
docker run -itd --net my-app-net -e constraint:node==node1 busybox
```



Other discovery backends

- Each discovery backend has a slightly different setup procedure
- Most options still require use of `swarm manage` and `swarm join` commands
- Can contribute your own discovery backend
- Reference examples at <https://docs.docker.com/swarm/discovery/>



TLS

- TLS can be enabled between
 - The Docker client and Swarm
 - Swarm and the Docker daemon on each node
- All daemon and client certificates must be signed using the same CA-certificate
- More at <https://docs.docker.com/swarm/install-manual/#tls>

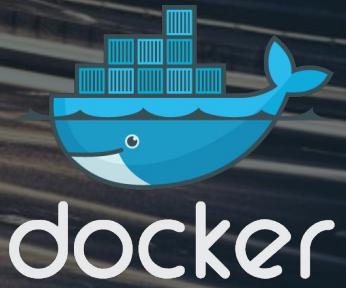


Docker Machine and Swarm

- Docker Machine can be used to provision a Swarm cluster
- Machine will create all the nodes for the cluster
- All nodes will be secured with TLS
- Works with any Docker Machine driver
- More at <https://docs.docker.com/machine/get-started-cloud/#using-docker-machine-with-docker-swarm>



Docker Compose



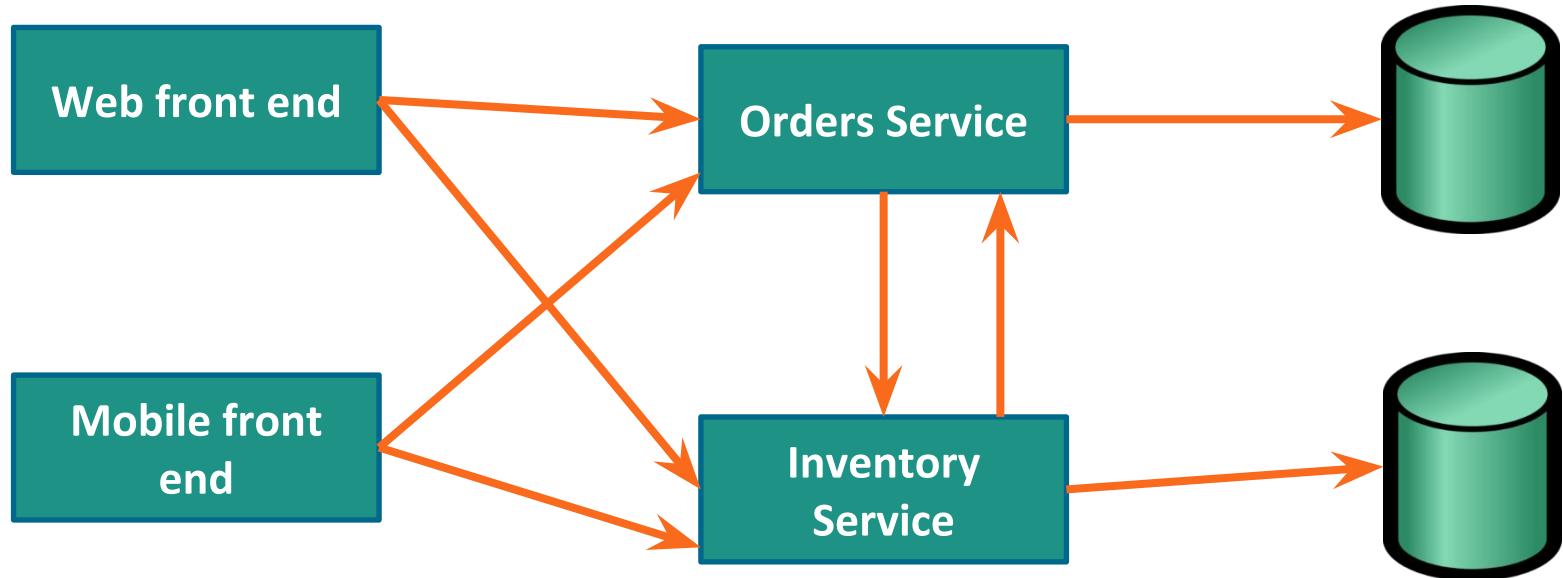
Module objectives

In this module we will:

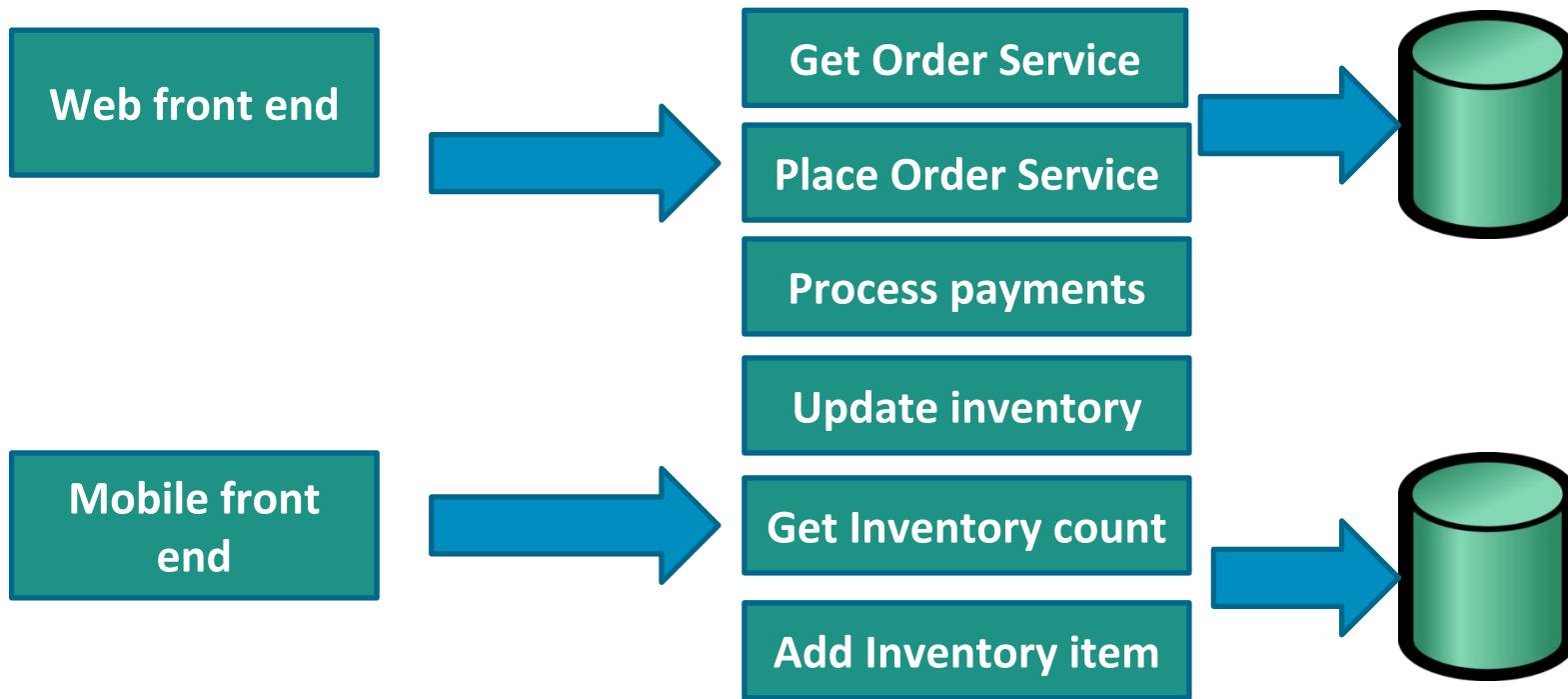
- Learn how to use Docker Compose to create and manage multi container applications
- Learn how to define a `docker-compose.yml` file
- Learn the key commands of `docker-compose`
- See how Docker Compose can be used with Docker Swarm



Recall simple micro service example



The reality



In a realistic micro service application

- There are lots of services, hence lots of components to run
- Do we really want to type `docker run ...` 50 times and specify 100's of links ?
- **Docker Compose** to the rescue



What is Compose?

*Docker **Compose** is a tool for creating and managing multi container applications*

- Containers are all defined in a single file called **docker-compose.yml**
- Each container runs a particular component / service of your application. For example:
 - Web front end
 - User authentication
 - Payments
 - Database
- Container links are defined
- Compose will spin up all your containers in a single command



Install Compose

- Check for the latest release at
<https://github.com/docker/compose/releases>
- Download the binary and place it into /usr/local/bin
- Rename binary to docker-compose
- Set the permission on the file
chmod +x /usr/local/bin/docker-compose

Note for MAC and Windows users: Docker Compose is included in Docker Toolbox



Install Docker Compose

1. Download the Docker Compose Linux binary from <https://github.com/docker/compose/releases>
2. Rename the binary file to docker-compose and place it into your path at /usr/local/bin
3. Set the correct file permission
`sudo chmod +x /usr/local/bin/docker-compose`
4. Verify the installation by running
`docker-compose --version`



Configuring the Compose yml file

- Defines the services that make up your application
- Each service contains instructions for building and running a container

service

Example

```
javaclient:  
  build: .  
  command: java HelloWorld  
  links:  
    - redis  
redis:  
  image: redis
```



Build instruction

- **Build** defines the path to the Dockerfile that will be used to build the image
- Container will be run using the image built
- Path to build can be relative path. Relative to the location of the yml file

Build image using
Dockerfile in current
directory

```
javaclient:  
  build: .  
  
orderservice:  
  build: /src/com/company/service
```



Image instruction

- **Image** defines the image that will be used to run the container
- Image can be local or remote
- Can specify tag or image ID
- All services must have either a build or image instruction

Use the latest redis
Image from Docker
Hub

```
javaclient:  
  image: johnnytu/myclient:1.0  
  
redis:  
  image: redis
```



Our compose example

- Two services
 - Java client
 - Redis
- The java client is a simple Java class that will connect to our Redis server and get the value of a key
- Code available at <https://github.com/johnny-tu/HelloRedis.git>



The Java code

```
import redis.clients.jedis.Jedis;

public class HelloRedis
{
    public static void main (String args [])
    {
        Jedis jedis = new Jedis("redisdb");

        while (true) {
            try {
                Thread.sleep(5000);
                System.out.println("Server is running: "+jedis.ping());
                String bookCount = jedis.get("books_count");
                System.out.println("books_count = " + bookCount);
            }
            catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```



The Dockerfile

```
FROM java:7
COPY /src /HelloRedis/src
COPY /lib /HelloRedis/lib

WORKDIR /HelloRedis
RUN javac -cp lib/jedis-2.1.0-sources.jar -d . \
src/HelloRedis.java
```



Links

- Same concept as container linking
- Specify <service name>:<alias>
- If no alias is specified, the service name will be used as the alias
- Creates an entry for the alias inside the container's /etc/hosts file

```
javaclient:  
  build: .  
  command: java HelloWorld  
  links:  
    - redis  
  
redis:  
  image: redis
```



Running your application

- Use `docker-compose up`
- Up command will
 - Build the image for each service
 - Create and start the containers
- Compose is smart enough to know which services to start first
 - Source containers are started up before recipients
- Containers can all run in the foreground or in detached mode



Running your application



View your containers

- Standard docker ps command not effective if you have many containers
- Use docker-compose ps
- This will only display the services that were launched from Compose as defined in the docker-compose.yml file
- Command needs to be run within the folder with the yml file

```
johnytu@docker-ubuntu:~/HelloRedis$ docker-compose ps
      Name           Command           State    Ports
-----  
helloredis_javaclient_1   java HelloRedis          Up
helloredis_redis_1        /entrypoint.sh redis-server   Up      6379/tcp
```



Container naming

- Container's launched by `docker-compose` have the following name structure
`<project name>_<service name>_<container number>`
- Project name is based on the base name of the current working directory unless otherwise specified
- Project name can be specified with the `-p` option or `COMPOSE_PROJECT_NAME` environment variable

```
johnnytu@docker-ubuntu:~/HelloRedis$ docker-compose ps
      Name           Command           State    Ports
-----
helloredis_javaclient_1   java HelloRedis
helloredis_redis_1        /entrypoint.sh redis-server   Up          6379/tcp
```



Project name **Service name**



Foreground vs detached mode

- In foreground mode, if one container stops, every other container defined and started by Compose will stop as well
- If we run in detached mode, this is not the case. Other containers will continue to run
- To launch all your containers in detached mode use
`docker-compose up -d`



Quick note on Compose commands

- Most commands for docker-compose can be run against a specific service
- If no service is specified, the command applies to all services defined in the docker-compose.yml file
- The service name is the name that you specified in the docker-compose.yml file, not the name of the container
- Full list of commands at <https://docs.docker.com/compose/cli/>



Start and stop services

- To stop a service

```
docker-compose stop <service name>
```

- To stop all services

```
docker-compose stop
```

- To start a service that has been stopped

```
docker-compose start <service name>
```

- To start all stopped services

```
docker-compose start
```



Remove services

- You can manually remove each service container with the `docker rm` command
- Or run `docker-compose rm` to delete all service containers that have been stopped
- Can specify a specific service to delete
`docker-compose rm <service name>`
- Use **-v** option to remove an associated volumes
`docker-compose rm -v <service name>`



View service container logs

- Use `docker-compose logs` command
- If a service is not specified, the aggregated log of all containers will be displayed
- Will automatically follow the log output
- Use `CTRL + C` to stop following



Scaling your services

- In a micro service architecture, we have the flexibility to scale a particular service to handle greater load without having to scale the entire application
- **Example:** If Orders is experiencing high traffic, scale the Order service by starting up more containers.
- Docker Compose has a convenient `scale` command
- Syntax

```
docker-compose scale <service name>=<number of instances>
```

Scale the orderservice up to 5 containers

```
docker-compose scale orderservice=5
```



Scaling up and down

- If the number of containers specified is greater than the current number for the service you specify, Docker Compose will start up more containers for that service until it reaches the number defined
- If the number specified is less than the current number of containers for that service, Docker Compose will remove the excess containers

Lets say we have 2 containers running for our orderservice. We want to scale to 5 containers.

```
docker-compose scale orderservice=5
```

The command has created an additional 3 containers. Now we no longer need that many and just need 1 container

```
docker-compose scale orderservice=1
```



Container naming with scaled services

- When a service has been scaled to multiple containers, running a docker-compose command against the service will apply to all containers
- For example
 - docker-compose logs javaclient
Will display the aggregated log output of all javaclient containers
- To run a command against just one individual container, use standard Docker commands and specify the container name
 - docker logs helloredis_javaclient_1

```
johnnytu@docker-ubuntu:~/HelloRedis$ docker-compose ps
      Name           Command       State    Ports
-----
helloredis_javaclient_1   java HelloRedis     Up
helloredis_javaclient_2   java HelloRedis     Up
helloredis_javaclient_3   java HelloRedis     Up
helloredis_javaclient_4   java HelloRedis     Up
```



Points to consider with scaling

- `docker-compose scale` command is for horizontal scaling (increasing the number of containers)
- Services have to be specially written to take advantage of this
- For example:
 - If we scaled multiple copies of a database container, will our other service containers automatically connect to them all?



Compose yml parameters

- So in our `docker-compose.yml` file we've seen the following parameters
 - build
 - image
 - links
- Some other parameters include
 - ports – for port mapping
 - volumes – for defining volumes
 - command – specifying which command to execute
- Full reference list at <https://docs.docker.com/compose/yml/>



Example parameters

```
web:  
  build: .  
  command: python app.py  
  ports:  
    - "5000:5000"  
  volumes:  
    - myvol:/code  
  volumes-from:  
    - mycontainer
```

Port mapping is specified as
<host port>:<container port>

Create a volume called “myvol” and mount it
in the /code folder of the container

Mount all volumes from the
mycontainer service



Yml parameters and Dockerfile instructions

- Most parameters in a `docker-compose.yml` file has an equivalent Dockerfile instruction
- Options that are already specified in the Dockerfile are respected by Docker Compose and do not need to be specified again
- Example
 - We expose port 8080 in our Dockerfile of a custom NGINX image
 - We want to build and run this as one of our services using Docker Compose
 - No need to use the `ports` parameter in the `yml` file as we have already defined it in the Dockerfile



Volume handling

- If your container uses volumes, when you restart your application by running `docker-compose up`, Docker Compose will create a new container, but re-use the volumes it was using previously.
- Handy for upgrading a stateful service, by pulling its new image and just restarting your stack



Docker Compose build

- `docker-compose build` command will build the images for your defined services
- Images are tagged as `<project name>_<service name>:latest`
- Run build, if you have changed the Dockerfile or source directory of any service
- Difference with `docker-compose up` ?
 - The `docker-compose up` command will build the service image, create the service and then start the service



Variable substitution

- Environment variables can be used in your yml configuration file
- Values are interpolated from the same variable on the host
- Variable syntax can be `$VARIABLE` or `${VARIABLE}`
- If the value of the variable on the host is empty, Compose will substitute an empty string

The variable `MYAPP_VERSION` needs to be set on the host. Its value is substituted into `$ (MYAPP_VERSION)`

```
webapp:  
  image: jtu/mywebapp:$ (MYAPP_VERSION)  
  ...  
  ...
```



Specifying another yml file

- Default Compose configuration file is `docker-compose.yml`
- We can specify another file as our Compose configuration file by using the `-f` option
- Example:
`docker-compose -f docker-compose.staging.yml`
- Allows you to have multiple configuration files. For example:
 - One for staging environment
 - One for production environment



Specifying multiple yml files

- When multiple Compose configuration files are specified using the `-f` option, the configuration of those files are combined.
- Example:

```
docker-compose -f docker-compose.yml \
    -f docker-compose.debug.yml
```

- Configuration is built in the order of the listed files
- If there are conflicts between certain fields in the files, subsequent files will override



Compose and Networking

- Networking support for Compose is available by specifying the `--x-networking` flag
- By default Compose creates a bridge network for your application and runs all the containers defined by the services in your configuration file in that network
- Network name is the project name
- Feature is still experimental



Using the bridge network with Compose

- Since all containers in a bridge network can lookup other containers using their container name, there is no need to define links in your configuration file
- Let's consider the following configuration example

```
docker-compose.yml
javaclient:
  build: .
redis:
  image: redis
```



Using the bridge network with Compose

- We will run `docker-compose --x-networking up -d` and inspect our containers

```
student@masterhost:~/HelloRedis$ docker-compose --x-networking up -d
Creating helloredis_redisdb_1
Creating helloredis_javaclient_1
student@masterhost:~/HelloRedis$ docker-compose ps
      Name           Command       State    Ports
-----  
helloredis_javaclient_1   java HelloRedis      Up
helloredis_redisdb_1     /entrypoint.sh redis-server  Up      6379/tcp
```



Using the bridge network with Compose

- Let's inspect our helloredis_javaclient_1 container

```
student@masterhost:~/HelloRedis$ docker inspect helloredis_javaclient_1
[
{
  "Networks": {
    "helloredis": {
      "EndpointID": "045fbabe6fd7ee7f63522de9b995dbf73d86210abdf8dbce2fa58f88153ab4f",
      "Gateway": "172.19.0.1",
      "IPAddress": "172.19.0.3",
      "IPPrefixLen": 16,
      "IPv6Gateway": "",
      "GlobalIPv6Address": "",
      "GlobalIPv6PrefixLen": 0,
      "MacAddress": "02:42:ac:13:00:03"
    }
  }
}
```



Using the bridge network with Compose

- Now let's check the `/etc/hosts` file of `helloredis_javaclient_1`
- Notice the entry for our `redis` container

```
student@masterhost:~/HelloRedis$ docker exec -it helloredis_javaclient_1 cat /etc/hosts
172.19.0.3      javaclient
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
172.19.0.2      helloredis_redisdb_1
172.19.0.2      helloredis_redisdb_1.helloredis
```



Custom container names with Compose

- By default, when using Docker Compose, container names are based on the project name and service name
- To specify a custom name we can use the `container_name` instruction
- **Note:** Compose cannot scale services with a custom container name

```
docker-compose.yml
javaclient:
  build: .
redis:
  image: redis
  container_name: redisdb
```



Using Compose and Swarm

- Run Docker Compose on a Swarm cluster simply by pointing your Docker client to Swarm
- Swarm will schedule the containers across different nodes if you are using the spread strategy
- **Note:** If you use links in your Compose configuration file, Swarm will schedule linked containers on the same node
- Containers running in different nodes cannot communicate with each other unless the nodes are part of an overlay network



Compose and overlay networks

- When specifying the `--x-networking` option, Compose uses a bridge network for the application
- When applied across multiple nodes in a Swarm cluster, this is not affective as containers won't be able to communicate
- Use the `--x-network-driver` option to specify an overlay network
 - Your nodes must be connected to a key / value store such as Consul etc...
 - Does not guarantee that applications in containers running on different nodes will be able to connect to each other

Example

```
docker-compose --x-networking \
    --x-network-driver=overlay up -d
```



Building vs Pulling images

- When using Docker Compose with Swarm, Compose does not have the ability to build an image across every Swarm node
- Compose will build the image in the node the container is scheduled on
- Affects ability to scale the service
- More effective approach is to build the image and push to Docker Hub, then have Compose pull the image into every Swarm node

Not ideal

```
javaclient:  
  build: .
```

More effective approach

```
javaclient:  
  image: trainingteam/hello-redis:1.0
```



Module summary

- Docker Compose makes it easier to manage micro service applications by making it easy to spin up and manage multiple containers
- Each service defined in the application is created as a container and can be scaled to multiple containers
- Docker Compose can create and run containers in a Swarm cluster



Orchestration Exercise

https://github.com/docker/dceu_tutorials/blob/master/2-orchestration.md

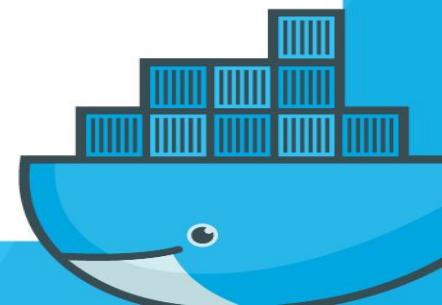


Tutum Training

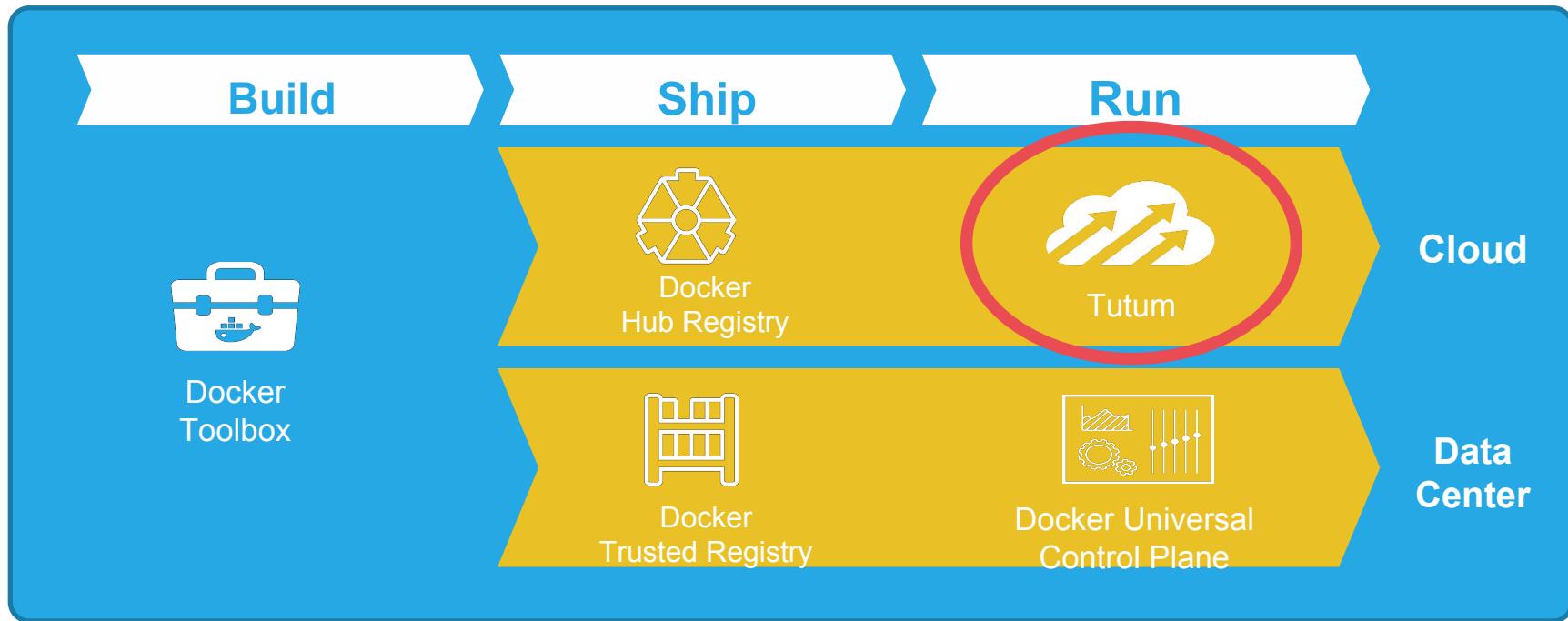


Agenda

- **Introductions**
- **Tutum overview (15 min)**
- **Hands on exercises (30 min)**
- **Q&A**



Docker Containers as a Service Platform



Tutum Key Features

- Delivered as SaaS

Currently in Beta, early access available, GA coming soon

- Declarative API (REST+WS) and Web UI+CLI

Applications: Stacks > Services > Containers

Infrastructure: Node Clusters > Nodes

Repositories

- Provisions Docker hosts in various cloud providers

AWS, Digital Ocean, Softlayer, Azure, Packet



Tutum Key Features

- **Build and Test repositories on any registry**

Tests using a simple compose file

Builds happen on the user nodes, not centrally

- **Managed overlay network and service discovery**

Automatically configures overlay network between nodes

DNS-based service discovery is provided as a service

- **Autoredeploy on push**

Automatically redeploys services when repos are updated



Overview



Exercises

- Add a “Bring Your Own Node”
- Deploy a “Hello World” service
- Load balance the “Hello World” service
- Deploy a load balanced “Hello World” stack



Exercises

<http://go.tutum.co/1luut8x>

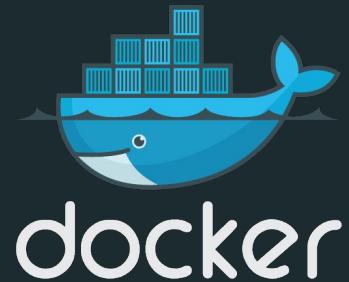


Thank you!

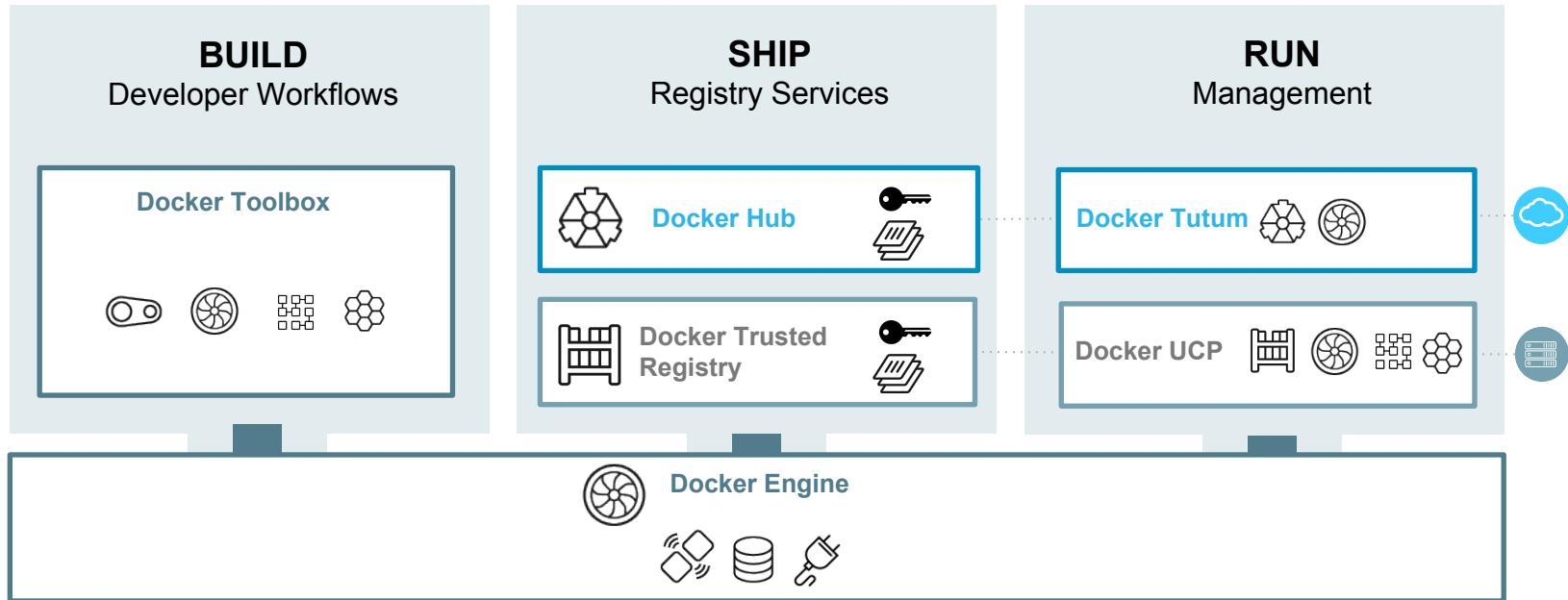


Docker Universal Control Plane: Customer Training Session

November 2015



The Docker Product Strategy



Docker Universal Control Plane Overview



Docker Universal Control Plane: The best way to deploy and manage Dockerized apps

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

For developers and IT ops to quickly and easily deploy and manage distributed apps

Docker native solution integrates Docker tools, API and leverages the broadest ecosystem

Fastest time to value with an easy to deploy and use solution



Enterprise ready

User Management

- LDAP/AD integration with Trusted Registry
- Role based access control (RBAC) to cluster, apps, containers, images

Some features
Coming Soon

Resource Management

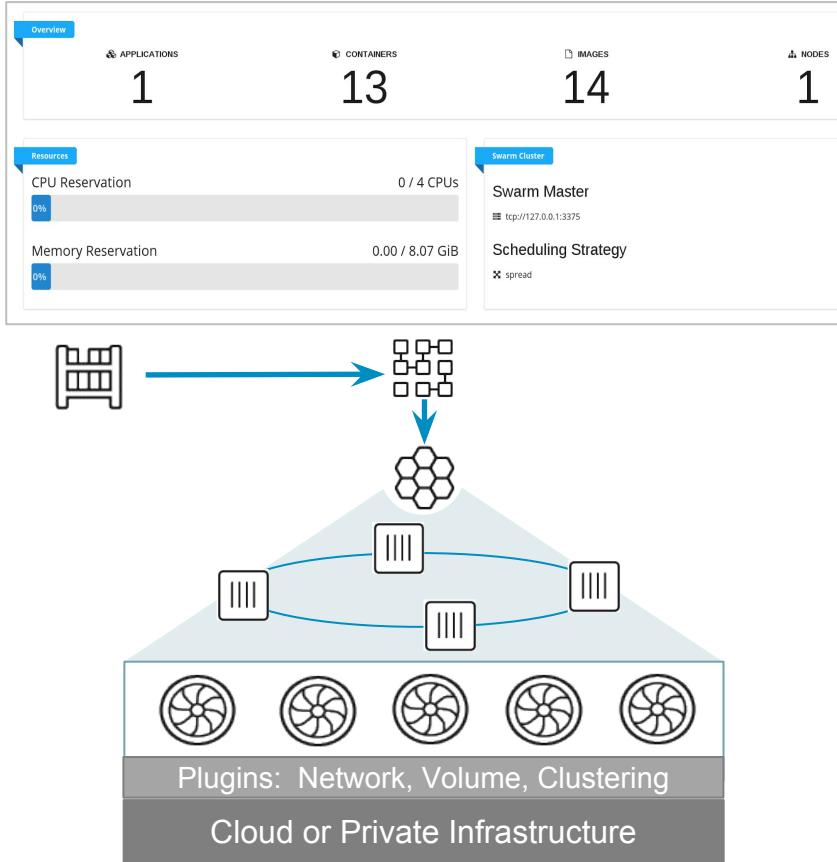
- Visibility into cluster, apps, containers, images, events with intuitive dashboards
- Manage clusters, images, network and volumes
- Manage apps and containers
- Monitoring and logging

Security & Compliance

- On-premise deployment
- Out of the box TLS
- LDAP/AD authentication
- User audit logs
- Out of the box HA



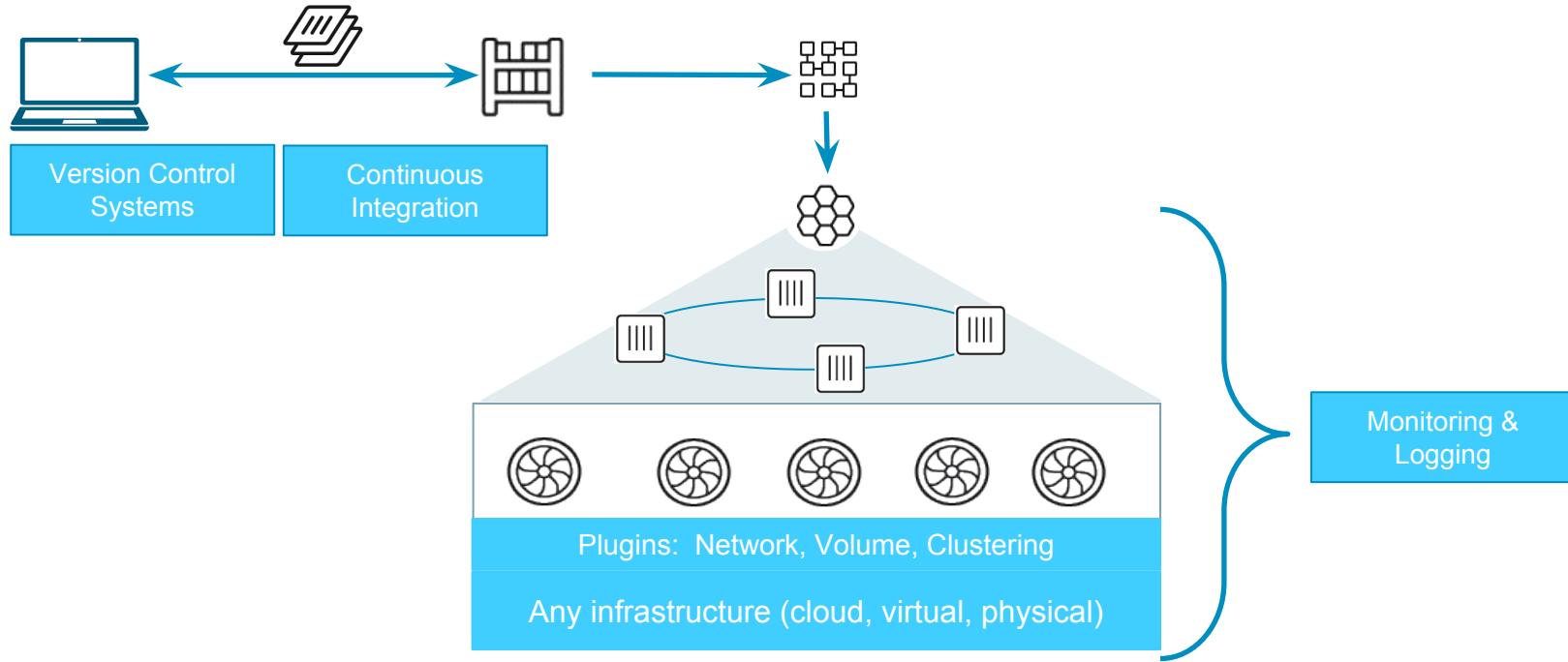
Docker native solution



- Built for Docker on Docker
- Fully Docker API compatible: Docker client, REST API
- Integrated Engine, Compose, Swarm and Trusted Registry
- Docker toolbox compatible
- Extensible to entire Docker ecosystem



Extensible to Docker ecosystem



Lab Time!



Before we start . . .

Pre-Requisites

You will need a laptop with SSH

Infrastructure

3 AWS EC2 Instances

- ducp-0: Runs the DUCP services
- ducp-1: Second node in our cluster
- ducp-3: Client node to be used for some exercises



Tasks

- Task 1: Install Docker Universal Control Plane on ducp-0
- Task 2: Add ducp-1 as a second managed node
- Task 3: Create an Nginx container
- Task 4: Download the Client bundle to ducp-2
- Task 5: Use Docker Compose from ducp-2 to stand up an app



Ok, let's get started

- Lab Guide is here: https://github.com/docker/ucp_lab
 - Feel free to clone or fork it
- When you're done, we'd love your (anonymous) feedback!

<http://bit.ly/1PxBamP>

- Real time feedback:

<http://bit.ly/1OdozSq>

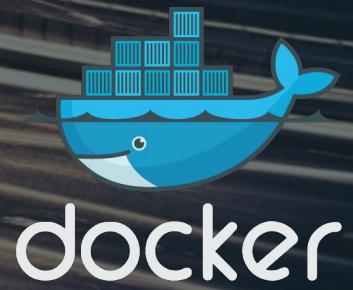
--fresh-install





THANK YOU

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>





THANK YOU