Advanced Docker Topics

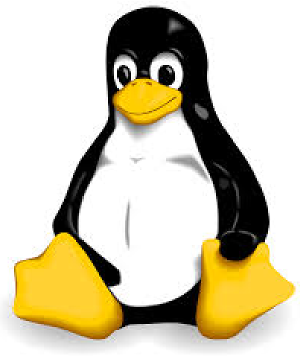
The open platform to build, ship and run any applications anywhere

SB4199C-001-1

Instructor Intro

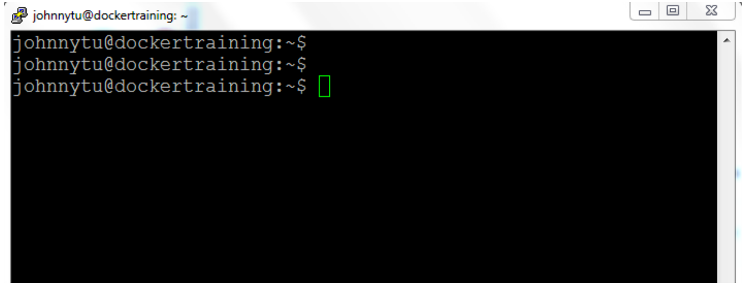
Session logistics and pre-requisites

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



Prerequisites

- Basic familiarity with Docker
- You will be provided with a Linux VM
- Be familiar with Linux command line



Your existing Docker knowledge

- You should be familiar with and know how to:
  - Pull images from Docker Hub
  - Create containers from images
  - Stop and start containers
  - Understand the lifecycle of a container
  - Run additional processes in a container (`docker exec` command)
  - Create images using a Dockerfile
  - Create volumes and attach them to containers
  - Run your existing applications in a container
  - Understand container networking and how to create a bridge network
  - Setup an automated build in Docker Hub
- If you have not done so already, consider taking the Introduction to Docker course first.

Your training environment

- You will be provided with four Ubuntu 14.04 instances
  - Master
  - Node 1
  - Node 2
  - Dtr-node
- **Your instructor will provide the login details**
- Unless otherwise specified, exercises will be done on the Master node
- Node 1 and node 2 will be used during the multi-host networking and Docker Swarm module
- Dtr-node will be used for installing Docker Trusted Registry
- Some optional exercises can be done on your own PC or Mac

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

Access your training environment

EXERCISE

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

Agenda

- Controlling the Docker daemon
- Security and TLS
- Multi host networking
- Docker Content Trust
- Setting up your own Registry
- Docker Trusted Registry
- Docker Machine
- Docker Swarm
- Building micro service applications
- Docker Compose

Module 1:
Controlling the Docker Daemon

Module objectives

In this module we will:

- Learn how to start and stop the Docker daemon when running it as a service
- Learn the various ways we can configure the daemon options
- Look at the Docker daemon log output and file
- Configure the Docker daemon to listen on a TCP socket
- Configure our Docker client to interact with a remote daemon

Controlling and configuring the Daemon

- The way we start / stop and configure the Docker daemon depends on a number of factors
  - Are we running it as a service ?
  - What Linux distribution
- `service` command vs `systemctl` command
- Running interactively in the foreground (`docker daemon …`)

Running as a service

For Ubuntu and Debian

- If you started Docker as a service, use `service` command to stop, start and restart the Docker daemon
  - `sudo service docker stop`
  - `sudo service docker start`
  - `sudo service docker restart`

Running as a service

For CentOS and Fedora

- We use the `systemctl` command to start and stop the Docker daemon
  - `systemctl start docker`
  - `systemctl stop docker`
  - `systemctl restart docker`
- You can actually use the `service` command as well but it will re-direct to `systemctl`

Running the daemon interactively

- If not running as a service, run Docker executable in daemon mode to start the daemon
  `sudo docker daemon &`
- If not running as a service, send a SIGTERM to the Docker process to stop it
  - Run `pidof docker` to find the Docker process PID
  - `sudo kill $(pidof docker)`

EX1.1 – Start and stop docker

EXERCISE

1. Stop the Docker daemon which should be running as a service
   `sudo service docker stop`
2. Run `docker version`. What do you notice on the output?
3. Now start the daemon again but this time use the docker client and specify the daemon command
   `sudo docker daemon &`
4. Run `docker version` again and verify that you can see the server version
5. Find the process ID of Docker and us it to stop the Docker daemon
   `sudo kill $(pidof docker)`
6. Start the daemon again using the `service` command

Docker daemon start up options

- How to configure the start up options will again depend on a few factors
  - Are you running as a service or running the binary directly (`docker daemon` …)
  - If running as a service are you using Ubuntu, Debian, CentOS or Fedora etc…
- If starting the Daemon from the Docker command you just specify the various options as a flag

```
sudo docker daemon [options] &
```

Docker daemon upstart configuration file

For Ubuntu and Debian

- Located in `/etc/default/docker`
- Use `DOCKER_OPTS` to control the startup options for the daemon when running as a service
- Restart the service for changes to take effect
  `sudo service docker restart`

Start daemon with log level of debug and allow connections to an insecure registry at the domain of myserver.org

```
DOCKER_OPTS="--log-level=debug --–insecure-registry=
myserver.org:5000"
```

Configuring the Daemon on CentOS

- CentOS uses `systemd` to run the Docker daemon
- Look for the `docker.service` file to see how Docker is started
- The `docker.service` file is found in either `/usr/lib/systemd/system` or `/etc/systemd/service`
- To be sure you can run `find / -name docker.service`

```
[root@docker-centos ~]# find / -name docker.service
/usr/lib/systemd/system/docker.service
```

docker.service file on CentOS

```
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.com
After=network.target

[Service]
Type=notify
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
Environment=GOTRACEBACK=crash
ExecStart=/usr/bin/docker -d $OPTIONS \
          $DOCKER_STORAGE_OPTIONS \
          $DOCKER_NETWORK_OPTIONS \
          $ADD_REGISTRY \
          $BLOCK_REGISTRY \
          $INSECURE_REGISTRY
LimitNOFILE=1048576
LimitNPROC=1048576
LimitCORE=infinity
MountFlags=slave

[Install]
WantedBy=multi-user.target
```

Docker configuration file for CentOS

- Notice how the `docker.service` file contains an `EnvironmentFile` property which points to `/etc/sysconfig/docker`
- This is the file where we can configure the Docker daemon’s start up options
- Start up options go in the `OPTIONS` flag

```
# /etc/sysconfig/docker
# Modify these options if you want to change the way the docker daemon runs
OPTIONS='--selinux-enabled'
DOCKER_CERT_PATH=/etc/docker

# If you want to add your own registry to be used for docker search and docker
# pull use the ADD_REGISTRY option to list a set of registries, each prepended
# with --add-registry flag. The first registry added will be the first registry
# searched.
#ADD_REGISTRY='--add-registry registry.access.redhat.com'
```

What are the Daemon options?

- What can we configure?
- Many options
  - Tell the Daemon to listen on a TCP socket
  - Specify a DNS server
  - Specify logging level
  - Enable debugging for logging
  - Add insecure registries
  - Enable and configure TLS
- For full reference list see https://docs.docker.com/reference/commandline/cli/#daemon

Docker daemon logging

- Start the docker daemon with `--log-level` parameter and specify the logging level
- Levels are (in order from most verbose to least):
  - Debug
  - Info
  - Warn
  - Error
  - Fatal

Run Docker daemon with debug log level (log written on terminal)

```
sudo docker daemon --log-level=debug
```

EX1.2 - Setup daemon logging

EXERCISE

1. Stop the Docker service or process that is currently running
   `sudo service docker stop` OR
   `sudo kill $(pidof docker)`
2. Start Docker daemon interactively and specify the debug logging level
   `sudo docker daemon --log-level=debug &`
3. Run a few Docker commands and observe the log output
4. Stop the Docker daemon
   `sudo kill $(pidof docker)`
5. Start it again and change the log level to info.
   `sudo docker daemon --log-level=info &`
6. Run some Docker commands and observe the log output

Configure logging in the upstart file

For Ubuntu and Debian

- Specify the `--log-level` option in the `DOCKER_OPTS` flag in `/etc/default/docker`
- You will have to restart the docker service for the changes to take effect
  `sudo service docker restart`
- Logs will be written to a file instead of the stdout
- Log file is `/var/log/upstart/docker.log`

Configuring in DOCKER_OPTS

```
DOCKER_OPTS="--log-level=debug"
```

Daemon log file on CentOS

- Since Docker runs via `systemd` on CentOS and Fedora the Docker daemon log is managed by `journald`
- Use the `journalctl` to view the log
- Just running `journalctl` will display the logs for everything managed through `systemd` so it's a good idea to filter the log so that only the entries from `docker.service` are displayed
- Recommended command
  `journalctl -u docker.service`

```
[root@docker-centos ~]# journalctl -f -u docker.service
-- Logs begin at Sun 2015-05-24 02:19:39 EDT. --
May 24 19:28:28 docker-centos docker[21501]: time="2015-05-24T19:28:28-04:00" level=debug ...01"
May 24 19:28:28 docker-centos docker[21501]: time="2015-05-24T19:28:28-04:00" level=debug ...8f"
May 24 19:28:28 docker-centos docker[21501]: time="2015-05-24T19:28:28-04:00" level=debug ...c3"
May 24 19:28:28 docker-centos docker[21501]: time="2015-05-24T19:28:28-04:00" level=debug ....."
May 24 19:28:28 docker-centos docker[21501]: time="2015-05-24T19:28:28-04:00" level=info m...e."
May 24 19:28:28 docker-centos docker[21501]: time="2015-05-24T19:28:28-04:00" level=info m...er"
```

EX1.3 – More logging

EXERCISE

1. Stop the Docker daemon
2. Open the `/etc/default/docker` file
3. Configure the `DOCKER_OPTS` flag to add the log level and specify info as the level
   `DOCKER_OPTS="--log-level=info"`
4. Start the Docker daemon using the service command
   `sudo service docker start`
5. Check the log file at `/var/log/upstart/docker.log` and notice the info level output
6. Run some simple command such as `docker ps` and check the log file again
7. Change the log level to warn and restart the service
8. Run some commands, check the log file again and notice the difference in output

Connecting to a remote daemon

- So far our Docker client and daemon have been on the same host
- What if we want to connect the client to a Docker daemon running on a different host?
- A few things we need to setup
  - First, the Docker daemon we want to connect to needs to be listening on a TCP socket
  - For security purposes we should use a HTTPS encrypted socket, which will require us to setup TLS (more on this later)
  - Then we point our client to the remote Daemon

Docker Daemon socket option

- The Docker daemon listens for remote API requests on three types of Socket
  - `unix`
  - `tcp`
  - `fd` (for Linux distributions using Systemd)
- The default socket is a `unix` domain socket created at `/var/run/docker.sock`
- This socket requires root permission

Error connecting to socket

- If you get the error message below, it typically means
  - The Docker daemon is not running
  - You do not have permission to make an API call to the docker daemon (i.e. you didn’t use sudo in your command or you are not in the docker group)
  - Your Docker client is trying to connect to the daemon using the unix socket but the daemon is not listening on it
  - You are not using TLS to connect to the daemon

```
ubuntu@node-0:~$ docker ps
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
```

EX1.4 – Sockets

1. Stop your Docker daemon
2. Try to run the following commands
   a) `docker`
   b) `docker version`
   c) `docker ps`
3. Notice the error message that appears on b) and c)

Listening on TCP socket

- To configure the Docker daemon to listen on a TCP socket, we start the daemon using the `--host` option and specify the TCP address and port
  - Can also use `-H`
- Be aware that by default the TCP socket is un-encrypted.
- For the address, you can specify an IP address to listen on or specify 0.0.0.0 to listen on all network interfaces.
- Port number should be 2375 for un-encrypted communication and 2376 for encrypted communication

Listening on TCP socket

Using docker command, listen on TCP socket for all network interfaces

```
docker daemon -H tcp://0.0.0.0:2375
```

Using docker command, listen on TCP socket on a particular IP address

```
docker daemon –H tcp://192.241.228.93:2375
```

Configuring via the upstart configuration file `/etc/default/docker`

```
DOCKER_OPTS="-H tcp://0.0.0.0:2375"
```

Listening on TCP socket

- CentOS Docker daemon configuration file example

```
# /etc/sysconfig/docker

# Modify these options if you want to change the way the docker daemon runs
OPTIONS='--selinux-enabled -H tcp://0.0.0.0:2375'
```

EX1.5 – Listen on TCP

EXERCISE

1. Open `/etc/default/docker`
2. Configure the `DOCKER_OPTS` variable to add the option for docker to listen on a tcp socket on any network interface
   `DOCKER_OPTS=“-H tcp://0.0.0.0:2375”`
3. Start the Docker service
4. Try to run a command such as `docker version` and `docker ps`
5. Notice you get the same error message as before (The error message from step 2 in Exercise 1.1
6. Why do you think this is?

Connect the client to the daemon

- By default the Docker client assumes the daemon is listening on a unix socket
- If the daemon is listening on a TCP socket, we have to configure the client to connect to a particular host
- Two methods
  - Use the `-H` flag on the docker command
  - Configure the `DOCKER_HOST` environment variable

Connect the client to the daemon

- Configuring the DOCKER_HOST environment variable is more effective as you only have to specify it once

Example of connect the client to a daemon listening on TCP using the docker command

```
docker –H tcp://localhost:2375
docker –H tcp://192.241.228.93:2375
```

Example using environment variable

```
export DOCKER_HOST="tcp://localhost:2375"
```

EX1.6 – Specify host on client

EXERCISE

1. Run a docker command with the `-H` option to connect to your daemon which is now listening on a TCP socket
   `docker -H tcp://localhost:2375 ps`
2. Now set the `DOCKER_HOST` environment variable to tcp://localhost:2375
   `export DOCKER_HOST=“tcp://localhost:2375”`
3. Run some docker commands without the `-H` flag. Verify that there are no error messages

Connecting the client to a remote daemon

- Make sure the Docker daemon on the remote host is listening on TCP
- Use the same method as before but specify the address of the remote host
- Once the client is connected to the remote daemon, all API calls from client commands will be run through the remote daemon
- Basically you will be performing actions on the remote host

Disconnecting the client

- If our client is connected to another Docker daemon on another host we can disconnect it by re-setting the `DOCKER_HOST` environment variable

```
export DOCKER_HOST=
```

OR

```
unset DOCKER_HOST
```

EX1.7 – Connect to remote daemon

EXERCISE

1. Pair up with the student next to you
2. Get the IP address of their AWS host
3. Connect your Docker client to the daemon running on the other students host
4. Run `docker ps -a` to see what containers the student has run
5. Run a new NGINX container in detached mode
6. Have to other student verify that the container is running on their host
7. Now disconnect the client from the other students host and swap roles. Let the other student connect their Docker client to your daemon

Listening on multiple sockets

- We can have our Docker daemon listening on both the Unix socket and TCP socket
- Just configure the `-H` option multiple times
- For Unix socket specify `unix:///var/run/docker.sock`

On docker command

```
docker daemon –H unix:///var/run/docker.sock \
          –H tcp://0.0.0.0:2375
```

On /etc/default/docker upstart configuration file

```
DOCKER_OPTS=“-H tcp://0.0.0.0:2375 \
        –H unix:///var/run/docker.sock”
```

EX1.8 – Multiple sockets

EXERCISE

1. From the end of the last exercise, your Docker client has been disconnected from the daemon on the remote host. Try and run a command such as `docker ps`. Notice the error since the client is trying to use the unix socket
2. Open `/etc/default/docker` and configure Docker to listen on both the TCP and unix socket
   ```
   DOCKER_OPTS="-H tcp://0.0.0.0:2375 \
       –H unix:///var/run/docker.sock"
   ```
3. Restart the Docker daemon
4. Now run `docker ps` again and verify that it works

Module summary

- The way to configure the Docker daemon varies by Linux distribution
- The Docker daemon listens for API requests on a unix socket by default
- We can configure the daemon to listen on a TCP socket
- We can connect to a daemon running on any host as long as the daemon is listening on TCP

Module 2: Security and TLS

Module objectives

In this module we will:

- Do a quick overview of the security considerations when running Docker
- Learn how to secure the Docker daemon with TLS

Linux containers and security

- Docker helps make applications safer as it provides a reduced set of default privileges and capabilities
- Namespaces provide an isolated view of the system. Each container has its own
  – IPC, network stack, root file system etc…
- Processes running in one container cannot see and effect processes in another container
- Control groups (Cgroups) isolate resource usage per container
  – Ensures that a compromised container won't bring down the entire host by exhausting resources

Quick security considerations

- Docker daemon needs to run as root
- Only ensure that trusted users can control the Docker daemon
  - Watch who you add to docker group
- If binding the daemon to a TCP socket, secure it with TLS
- Use Linux hardening solution
  - Apparmor
  - SELinux
  - GRSEC

Transport Layer Security

- Evolution of SSL
- The protocol that secures websites with https URLs.
- Uses Public Key Cryptography to encrypt connections.
- Keys are signed with Certificates which are maintained by a trusted party.
- These Certificates indicate that a trusted party believes the server is who it says it is.
- Each transaction is therefore encrypted *and* authenticated.

Using TLS for Docker

- Docker provides mechanisms to authenticate both the server and the client to each other.
- Provides strong authentication, authorization and encryption for any API connection over the network.
- Client keys can be distributed to authorized clients
- **Before we begin**
  - Make sure you have OpenSSL 1.0.1 installed
  - Create a folder to store your keys and make sure the folder is protected (use `chmod 700` to set the permissions)

Process overview for setting up TLS

- Create the Certificate Authority (CA)
  - Need a CA private key and certificate
- Setup the server private key
- Create a certificate signing request (CSR) for the server
- Sign the server key with the CSR against our CA
- Create a client private key and CSR
- Sign the client key with the CSR against our CA
- Run the Docker daemon with TLS enabled and specify the location of the CA private key, server certificate and server key
  - And configure it to listen on TCP
- Point the Docker client to the TCP address of the daemon and specify the location of the client certificate and key as the CA private key

Create the Certificate Authority

- We need the certificate authority to sign our server and client keys later on

Create the CA private key. You be prompted for a passphrase. Make sure you remember it

```
openssl genrsa -aes256 -out ca-key.pem 2048
```

Create the CA certificate (public key)

```
openssl req -new -x509 -days 365 \
            -key ca-key.pem -sha256 -out ca.pem
```

EX2.1 – Create the CA

EXERCISE

1. First we need to setup our folder to store all the keys. Create a folder called `docker-ca`
2. Run `chmod 0700 docker-ca` to set the correct permission on the folder
3. Go into the folder
4. Now create the Certificate Authority private key
   ```
   openssl genrsa -aes256 -out ca-key.pem 2048
   ```
5. Using the CA private key, create the CA certificate
   ```
   openssl req -new -x509 -days 365 \
             -key ca-key.pem -sha256 -out ca.pem
   ```

Setup the server key and CSR

- The certificate signing request (CSR) is needed so we can sign our server key.
- When creating the CSR, make sure you specify the hostname of the machine that your Docker daemon runs on in the CN attribute

Create the server private key

```
openssl genrsa -out server-key.pem 2048
```

Create the CSR. Notice the `CN=<host name>`. This is the DNS name of the host machine the Docker daemon is running on

```
openssl req -subj "/CN=<host name>" \
             -new -key server-key.pem -out server.csr
```

EX2.2 – Setup server key

EXERCISE

1. Create the server private key
   `openssl genrsa -out server-key.pem 2048`
2. Create the certificate signing request (CSR) and specify the IP address of the machine that the Docker daemon is running on
   ```
   openssl req -subj "/CN=<host name>" \
             -new -key server-key.pem -out server.csr
   ```
3. So far you should have the following files in your `docker-ca` folder
   a) CA private key – `ca-key.pem`
   b) CA certificate – `ca.pem`
   c) Server private key – `server-key.pem`
   d) Server CSR – `server.csr`

Sign the server key

- Before we sign our server key we will define a certificate extension to specify the `subjectAltName`
- The `subjectAltName` allows us to specify things such as the IP addresses we will allows connections on.

Create the certificate extension file with `subjectAltName` and allow connections over the IP addresses specified

```
echo subjectAltName = IP:10.10.10.20,IP:127.0.0.1 >
extfile.cnf
```

Sign the server key

- We now sign the server key using the Certificate Authority we created
- Specify the certificate extension file (`extfile.cnf`) as well

```
openssl x509 -req \
            -days 365 \
            -in server.csr -CA ca.pem \
            -CAkey ca-key.pem \
            -CAcreateserial \
            -out server-cert.pem \
            -extfile extfile.cnf
```

EX2.3 – Sign the server key

EXERCISE

1. Create a file called `extfile.cnf`
2. Open the file and using the `subjectAltName` extension to specify the IP addresses to allow connections to. You will need to specify the IP of your Docker daemon host and also 127.0.0.1
   ```
   subjectAltName = IP:<host IP>,IP:127.0.0.1
   ```
3. Sign the server key
   ```
   openssl x509 -req \
             -days 365 \
             -in server.csr -CA ca.pem \
             -CAkey ca-key.pem \
             -CAcreateserial \
             -out server-cert.pem \
             -extfile extfile.cnf
   ```

Create client keys

First we create the clients private key

```
openssl genrsa -out client-key.pem 2048
```

Then we create the client certificate signing request

```
openssl req -subj '/CN=client' \
            -new \
            -key client-key.pem \
            -out client.csr \
```

EX2.4 – Create client key

EXERCISE

1. Create the client private key. Call it `client-key.pem` to make is easy to distinguish from our server key
   ```
   openssl genrsa -out client-key.pem 2048
   ```
2. Create the client CSR using the following command
   ```
   openssl req -subj '/CN=client' \
            -new \
            -key client-key.pem \
            -out client.csr \
   ```

Sign client keys

We need an extensions config file with the extendedKeyUsage extension in order to make the key suitable for client authentication

```
echo extendedKeyUsage = clientAuth > extfile.cnf
```

Now we can sign our client public key

```
openssl x509 -req -days 365 \
             -in client.csr \
             -CA ca.pem \
             -CAkey ca-key.pem \
             -CAcreateserial \
             -out client-cert.pem \
             -extfile extfile.cnf
```

EX2.5 – Sign the client key

EXERCISE

1. Create the extension file to make the key suitable for client authentication
   ```
   echo extendedKeyUsage = clientAuth > extfile.cnf
   ```
2. Sign the clients public key
   ```
   openssl x509 -req -days 365 \
             -in client.csr \
             -CA ca.pem \
             -CAkey ca-key.pem \
             -CAcreateserial \
             -out client-cert.pem \
             -extfile extfile.cnf
   ```

Enable TLS on the Docker daemon

Two options

1. Run `docker daemon` and specify the following flags
```
--tlsverify
--tlscacert=<path to ca cert>
--tlscert=<path to server certificate>
--tlskey=<path to server key>
-H=0.0.0.0:2376
```
2. Or we can put those flags into the Daemon configuration file and run the daemon as a service (`/etc/default/docker` for Ubuntu)

Protect our certificates and keys

- For the CA private key, server private key and client private key (`ca-key.pem`, `server-key.pem` and `client-key.pem`) you want to make the files readable only to yourself

```
chmod -v 0400 ca-key.pem client-key.pem \
                server-key.pem
```

- For the certificates you will want to remove write access

```
chmod -v 0444 ca.pem server-cert.pem client-cert.pem
```

- It is also recommended that you move the server certificate, server private key and CA private key into a system folder such as `/etc/docker`

EX2.6 – Securing the keys

EXERCISE

1. In the docker-ca folder, make the CA, server and client private keys readable only to yourself.
   ```
   chmod -v 0400 ca-key.pem client-key.pem \
                              server-key.pem
   ```
2. Remove write access to all certificates.
   ```
   chmod -v 0444 ca.pem server-cert.pem client-cert.pem
   ```
3. Create folder `/etc/docker` if it does not already exist.
4. Make yourself the owner of the `/etc/docker` folder.
   ```
   sudo chown <username>:docker /etc/docker
   ```

EX2.6 – Securing keys (cont'd)

5. Set read, write and execution permissions for yourself only on the `/etc/docker` folder.
   ```
   sudo chmod 700 /etc/docker
   ```
6. Copy the CA key, server key and server certificate to the `/etc/docker` folder.
   ```
   sudo cp ~/docker-ca/{ca,server-key,server-cert}.pem
   /etc/docker
   ```

EX2.7 – Enable TLS on the daemon

EXERCISE

1. Open the /etc/default/docker file
2. Change the DOCKER_OPTS variable to the following (all on one line):
```
DOCKER_OPTS="-H tcp://0.0.0.0:2376 --tlsverify
--tlscacert=/etc/docker/ca.pem
--tlscert=/etc/docker/server-cert.pem
--tlskey=/etc/docker/server-key.pem”
```
3. Restart the docker service
```
sudo service docker restart
```

Specifying TLS on the client

- Now that the Docker daemon has TLS enabled, when we use the client, we need to specify to enable TLS as well and specify our client certificate and key
- Run `docker` and specify the following flags and then the command

```
--tlsverify
--tlscacert=<path to ca cert>
--tlscert=<path to client certificate>
--tlskey=<path to client key>
-H=<server url>:2376
```

Specifying TLS on the client

Example command of running the docker client with the TLS flags

```
docker --tlsverify \
       --tlscacert=ca.pem \
       --tlscert=client-cert.pem \
       --tlskey=client-key.pem \
       -H=tcp://127.0.0.1 \
      ps –a
```

EX2.8 – Run the client

EXERCISE

1. Go into your `docker-ca` folder and run:
```
docker --tlsverify \
       --tlscacert=ca.pem \
       --tlscert=client-cert.pem \
       --tlskey=client-key.pem \
        -H tcp://127.0.0.1:2376 \
         ps
```
2. Verify that the client is able to talk to the daemon. Try a few more commands

Make life more convenient

- It is not very convenient to have to specify all those TLS flags everytime we want to run `docker` command
- To get around this, we can place our client key and certificate along with the CA key into a hidden folder called `.docker`. This folder resides in our home directory.
- However the file names must be `ca.pem, cert.pem` and `key.pem`
- After this whenever you run the `docker` command, the client knows to supply the key and certificate as part of the request and you will only need the `--tlsverify and -H` options
  `docker --tlsverify -H 127.0.0.1:2376 ps -a`

Environment variables

- To make things even simpler, we can set the `DOCKER_HOST` environment variable to the TCP address the daemon is listening on
  `export DOCKER_HOST=“tcp://<ip address>:2376”`
- We can also set the `DOCKER_TLS_VERIFY` variable to 1, which will tell the client to pass the `--tlsverify` flag on every request
  `export DOCKER_TLS_VERIFY=1`

EX2.9 – Set environment variables

EXERCISE

1. Create the .docker folder in your home directory
   `mkdir ~/.docker`
2. In your docker-ca folder Copy `ca.pem`, `client-cert.pem` and `client-key.pem` into the `.docker` folder
3. Cd into `~/.docker` and rename `client-cert.pem` to `cert.pem`
4. Rename `client-key.pem` to `key.pem`
5. Set the `DOCKER_HOST` environment variable to 127.0.0.1
   `export DOCKER_HOST="tcp://127.0.0.1:2376"`
6. Set the DOCKER_TLS_VERIFY variable to 1
   `export DOCKER_TLS_VERIFY=1`

EX2.9 (cont'd)

EXERCISE

7. Now run a few `docker` commands with specifying any flags and verify that it works
   `docker ps`
   `docker images`
8. Clean up your setup by removing all TLS settings
   a) Open `/etc/default/docker` and remove all `--tls` options from `DOCKER_OPTS`
   b) Change `–H` option to `tcp://0.0.0.0:2375` and add another `–H` option with value `unix:///var/run/docker.sock`
   c) Restart the Docker daemon
   d) Run `unset DOCKER_HOST` and `unset DOCKER_TLS_VERIFY`

Module 3:
Multi-host Networking

Module objectives

- Explain the requirements of setting up a multi-host network
- Configure a multi-host networking across two nodes
- Run containers on different hosts which are linked together by the multi-host network

Multi-host networking

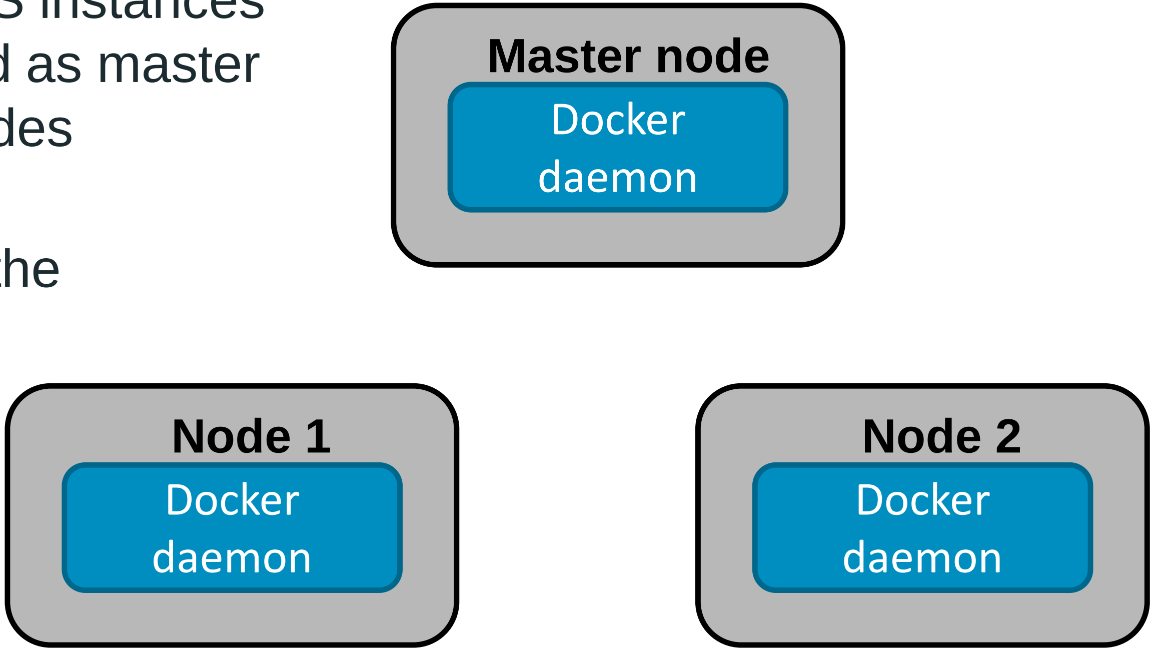
- Containers running on different hosts cannot communicate with each other without mapping their TCP ports to the host's TCP ports
- Multi-host networking allows these containers to communicate without requiring port mapping
- The Docker Engine supports multi host networking natively out of the box via 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 Master node



Step 1 - Setup key-value store

Perform this on your Master Node

- 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 \
      progrium/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 progrium/consul -server -bootstrap
2c54ff68658d937aee4161735c65de43aef71ba31cdc24645161e65d8d3aa175
student@dockerhost:~$ docker ps
CONTAINER ID        IMAGE              ...    STATUS             PORTS                              NAMES
2c54ff68658d        progrium/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 master node created in step 1
- **Note:** You will need to know the IP address of your Master Node

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 <Master Node IP> with the IP address of your Master Node
  - 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://<Master Node 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

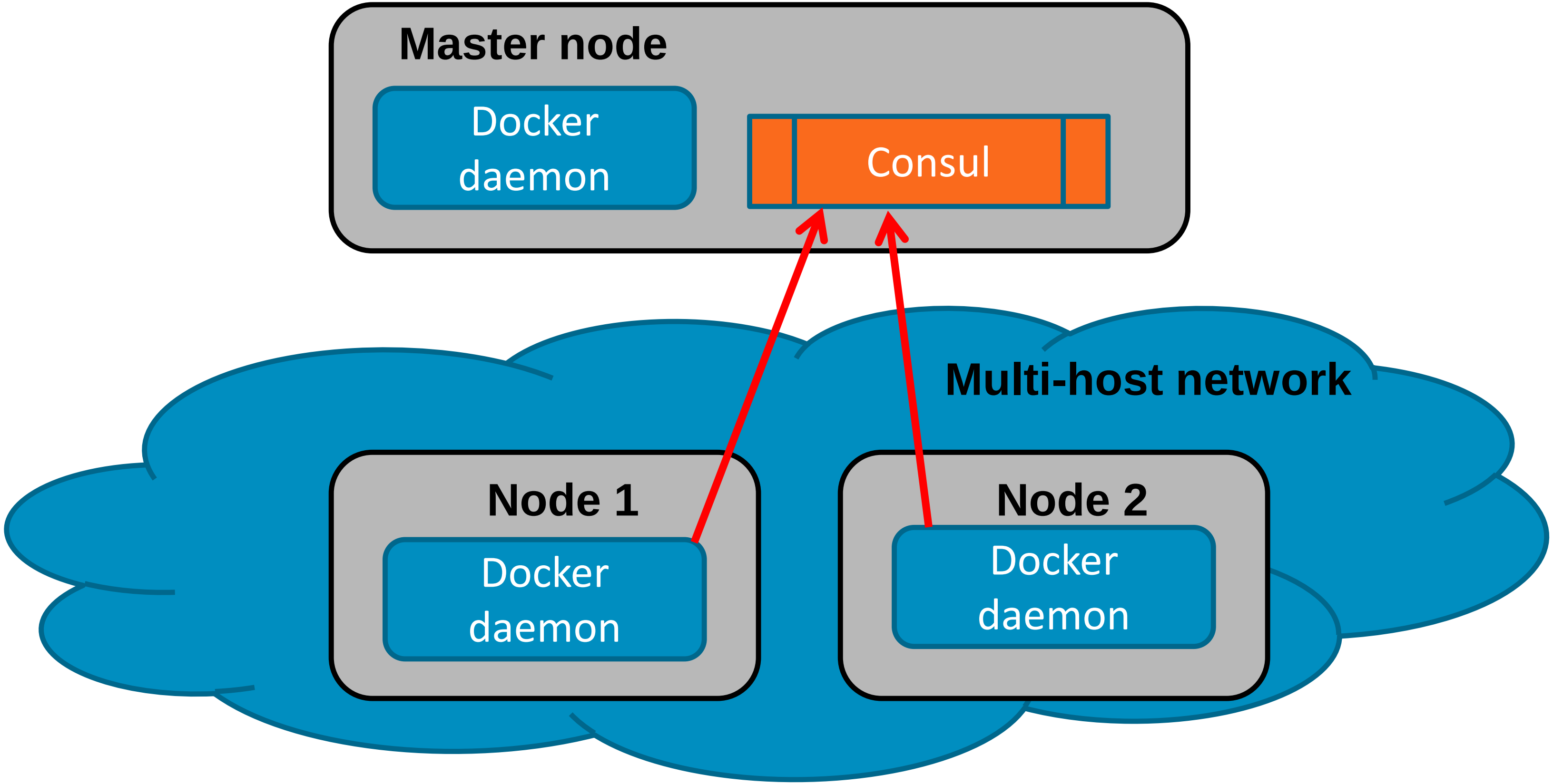
- We will create an overlay network called `multinet`
- It will be configured with the 10.10.10.0/24 subnet
- Run the command
  `docker network create -d overlay --subnet 10.10.10.0/24 multinet`

```
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



EX3.1 – Setup the multi-host network

EXERCISE

1. Make sure you have no containers running on any of your nodes
2. Make sure you have the IP address of your Master Node
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

- The docker daemon contains an embedded DNS server
- Containers must run with a name (using the `--name` option). This maps to the IP address on the network the container is connected to.
- When a container is added to a multi-host network, all other hosts will be able to discover it via the DNS server
- Container may have any number of aliases on a network
- Containers may have different aliases on different networks, set using the `--alias` option on `network connect`

Container discovery

- At this stage, we have setup a container on Node1 called c1
- Let's run another container on Node2 and ping node 1 using the name

```
root@node2:~# docker run -d --name nginx --net multinet
nginx
E7576fd798aac025bc73c395011adc73007fe184c1646c645ed99af
acffc177b
root@node2:~# docker exec nginx ping c1
Pinging c1 [10.10.10.2] with 32 bytes of data:
Reply from 10.10.10.2: bytes=32 time=<1ms TTL=56
```

EX3.2 – Run container on multi-host network

EXERCISE

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. Verify that you can ping your `c1` container
   `docker exec nginx ping c1`

EX3.2 – (cont'd)

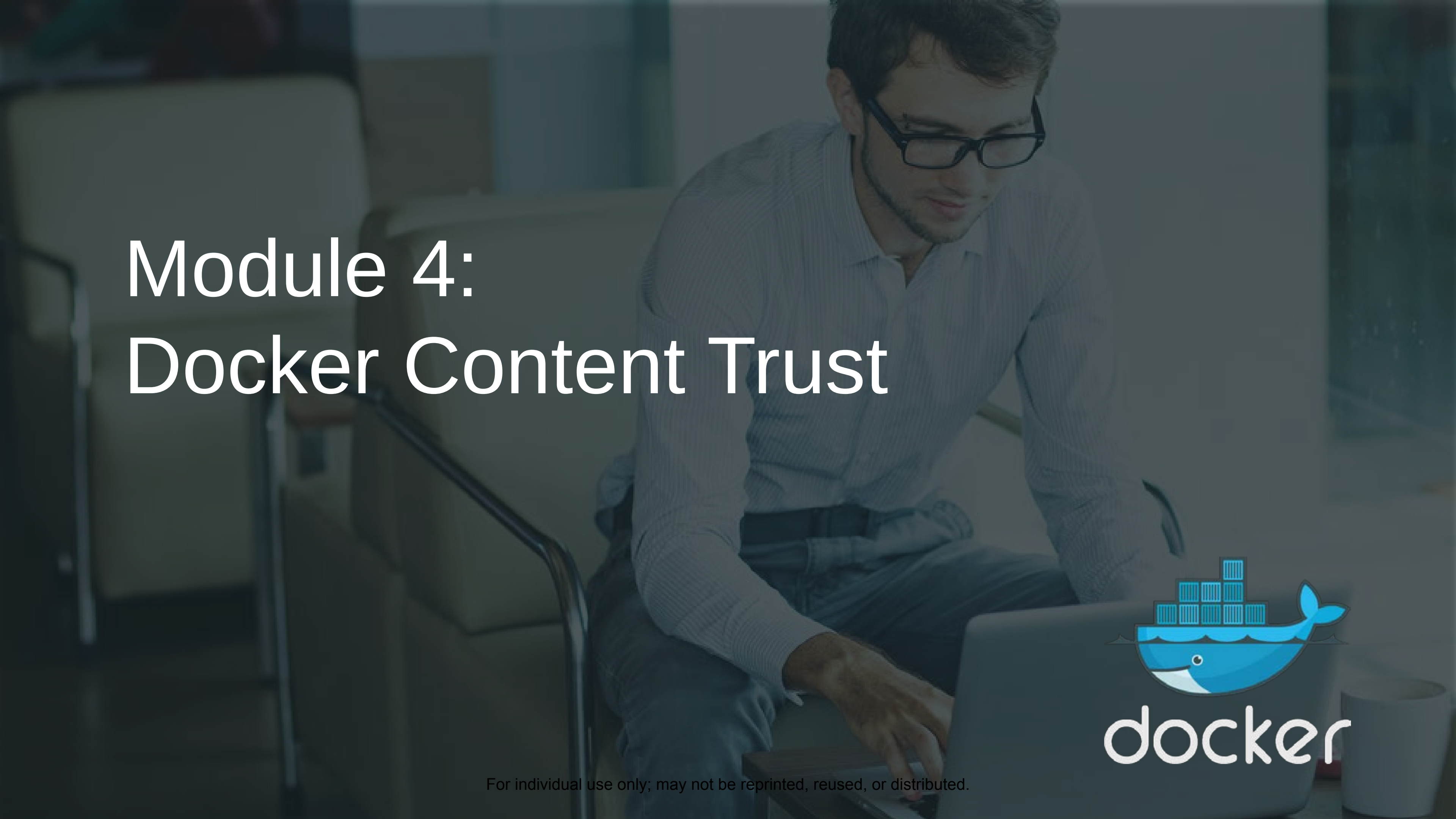
EXERCISE

Using Node1

6. Verify that you can `ping` the nginx container
   `docker exec c1 ping nginx`
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`

Module Summary

- An overlay (multi-host) network requires a key/value store
- Containers added to a multi-host network are discoverable by other containers, as long as the container name/alias has been specified
- Containers on different hosts can communicate with each other without exposing any ports if the hosts are part of the same overlay network



Module 4: 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

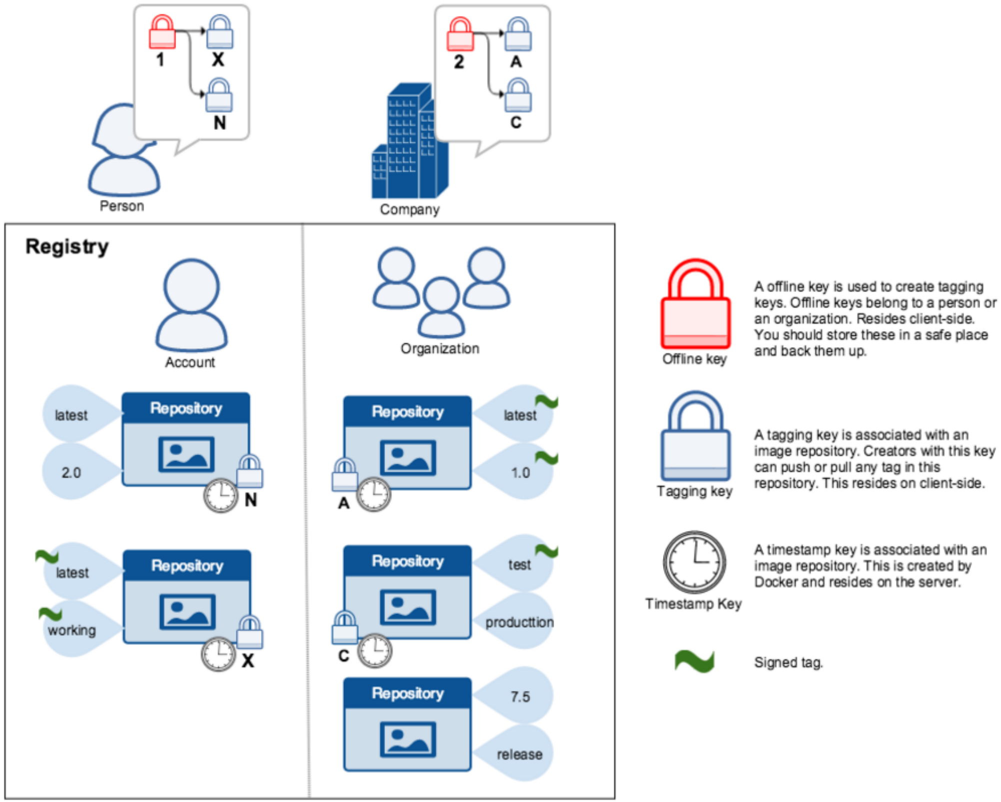
Docker Content Trust

- 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
```

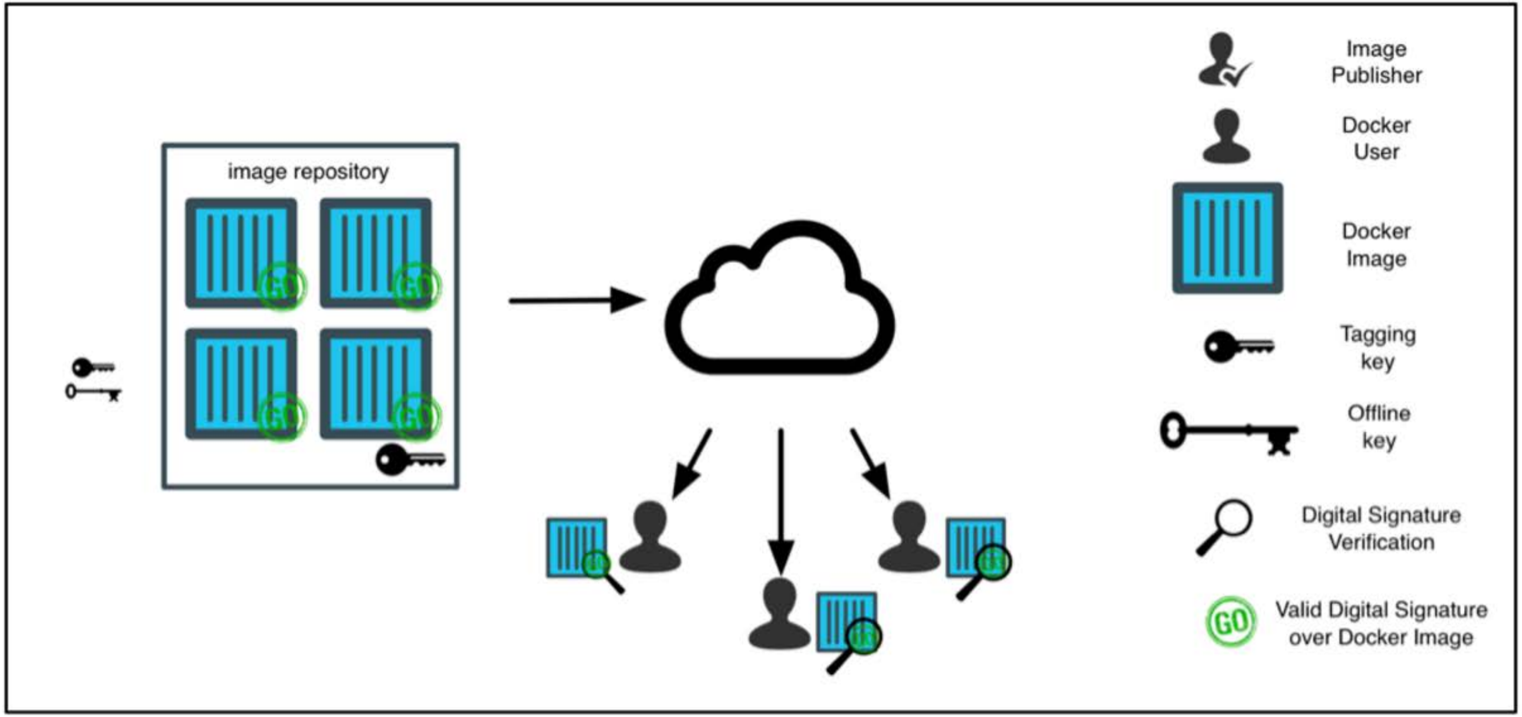
Pushing a signed image

```
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
```

EX4.1 – Using signed images

EXERCISE

1. Login to your Docker Hub account on your terminal
   `docker login`
2. Pull the latest Ubuntu image
   `docker pull ubuntu:latest`
3. Create a new tag for the image using your Docker Hub username. Call your new repository trustedubuntu
   `docker tag ubuntu:latest <username>/trustedubuntu:latest`
4. Push the image to Docker Hub
   `docker push <username>/trustedubuntu:latest`
5. Login to your Docker Hub account on a browser and check to make sure you can find your new Repository and Tag

EX4.1 – (cont'd)

EXERCISE

6. Enable content trust by setting the environment variable
   `export DOCKER_CONTENT_TRUST=1`
7. Try and run a container using your `trustedubuntu` image
   `docker run -it <username>/trustedubuntu:latest`
8. Notice the error message saying:
   "`no trust data available`"

   This is because the image we are trying to run the container from is not signed.

EX4.2 – Sign an image

EXERCISE

1. At this stage, content trust should be enabled. Tag your `trustedubuntu` image again as 1.0
   `docker tag <username>/trustedubuntu:latest <username>/trustedubuntu:1.0`
2. Push your new image tag to Docker Hub. You will be prompted for a root key passphrase and repository key passphrase. Pick a passphrase and make sure you remember it
   `docker push <username>/trustedubuntu:1.0`
3. Change directory into `~/.docker/trust/private/root_keys`. Verify you can see your root key in here
4. Change directory into `~/.docker/trust/private/tuf_keys/docker.io`. You should see a folder named after your Docker Hub username and underneath that another folder called `trustedubuntu`.
5. Change directory into the `trustedubuntu` folder in step 4
6. You should see your repository keys in this folder
7. Run a container using your newly signed `trustedubuntu:1.0` image
   `docker run -it <username>/trustedubuntu:1.0`

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

EX4.3 – Signed and unsigned tags

EXERCISE

Using your Master Node

1. Turn off content trust
   `unset DOCKER_CONTENT_TRUST`
2. Run a container using the `trustedubuntu:1.0` image from Exercise 4.2. Get terminal access into the container
   `docker run -it <username>/trustedubuntu:1.0 bash`
3. Add a file called `unsigned.txt`
   `touch unsigned.txt`
4. Exit the container
5. Commit the container as a new **unsigned** image with the 1.0 tag.
   `docker commit <container id> <username>/trustedubuntu:1.0`
6. Push the unsigned image to Docker Hub
   `docker push <username>/trustedubuntu:1.0`

EX4.3 – (cont'd)

EXERCISE

Switch over to your Node 1 instance. At this stage, content trust should be disabled

7. Run a container with a `bash` terminal using the `trustedubuntu:1.0` image
   `docker run -it <username>/trustedubuntu:1.0 bash`
8. On the container terminal, check to see that you have the `unsigned.txt` file underneath the / folder
9. Exit the container
10. Run another container using the same image but this time run it with content trust enabled
    `docker run -it --disable-content-trust=false <username>/trustedubuntu:1.0 bash`
11. Try and find the `unsigned.txt` file in the new container
12. Notice it is not present because our signed image is currently slightly older than the unsigned image and we did not create that file in it

EX4.4 - Cleanup

EXERCISE

1. Disable content trust on your Master Node and Node 1 in preparation for future exercises
   `unset DOCKER_CONTENT_TRUST`

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 …
```

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

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

Module 5:
Run Your Own Registry Server

Module objectives

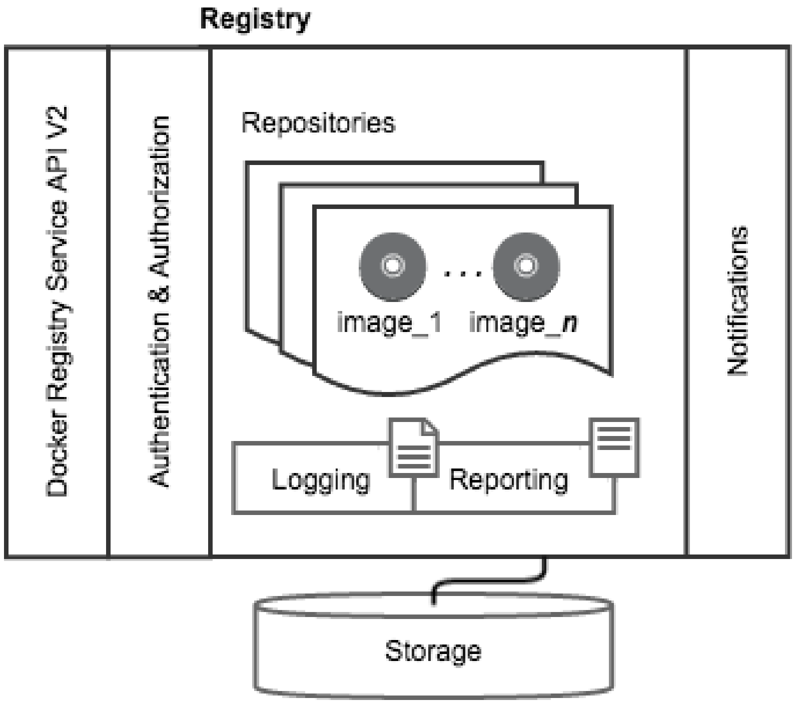
In this module we will:

- Learn how to run our own registry server
- Push and pull images into our registry server
- Explain the difference between a secure and insecure registry
- Outline the factors to take into account when running registry server for production use

Registry server

- Run your own registry server to store and distribute images instead of using Docker Hub
- Multiple options
  - Run registry server using container
  - Docker Trusted Registry
- **Two versions**
  - Registry v1.0 for Docker 1.5 and below (deprecated)
  - Registry v2.0 for Docker 1.6 and above

Registry server features



Registry server features

- Configurable storage. Store your images on
  - Local disk
  - Amazon S3
  - Microsoft Azure
- Webhooks to kick off a CI build or send notifications to certain people when images have been pushed
- Secure access to your images via TLS

Public or private

- You can choose to run a registry server that is publicly available for people to pull images from and push images to
- You can put it behind the firewall and make it available only to internal company staff, contractors etc…

Setting up a registry server

- Two methods
  - Use the official **registry** image at https://registry.hub.docker.com/u/library/registry/
  - Download the distribution source and build your own custom registry image
- Official image contains a pre-configured version of registry v2.0
- Official image is meant to evaluation purposes as it's default configuration is not suitable for production use
  - No TLS enabled

Run a new container using the registry image

```
docker run -d -p 5000:5000 registry:2.0
```

EX5.1 – Setup a registry server

EXERCISE

1. Setup a registry server by running the official registry image. Map port 5000 on the container to port 5000 on your host
   `docker run –d –p 5000:5000 registry:2.0`
2. To verify that the server is up and running you can try and make a basic API call to `http://<server url>:5000/v2/.` You should get a HTTP 200 response. On your terminal run
   `curl -i http://localhost:5000/v2/`
3. Verify that you get a HTTP 200 response

Push image to the registry

- First tag the image with host IP or domain of the registry server, then run `docker push`

Tag image and specify the registry host

```
docker tag <image id>  myserver.net:5000/my-app:1.0
```

Push image to registry

```
docker push myserver.net:5000/my-app:1.0
```

Example of using image tag with a user or group name

```
docker tag <image id> myserver.net:5000/johnnytu/my-app:1.0
docker push myserver.net:5000/johnnytu/my-app:1.0
```

EX5.2 – Push images

EXERCISE

1. Pull the latest busybox image from Docker Hub
   `docker pull busybox`
2. Tag the image as `mybusybox` and specify the registry host URL. Also, specify your name as part of the repository. Specify a 1.0 tag. For example:
   `docker tag busybox localhost:5000/johnnytu/mybusybox:1.0`
3. Push the image to your registry server
   `docker push localhost:5000/johnnytu/mybusybox:1.0`
4. Tag the local busybox image again, this time use 1.1
   `docker tag busybox localhost:5000/johnnytu/mybusybox:1.1`
5. Push the image from 4) into the registry server
   `docker push localhost:5000/johnnytu/mybusybox:1.1`
6. Delete the local busybox images
   `docker rmi busybox`

Check repository tags on the registry

- To see what tags are available for a particular image repository we make an API call to the following URL
<registry host>:<port>/v2/<repo name>/tags/list
- For example: to get the tags for the busybox images we just pushed to our registry we would hit
http://<server url>:5000/v2/johnnytu/mybusybox/tags/list

```
$ curl http://localhost:5000/v2/johnnytu/mybusybox/tags/list
{"name":"johnnytu/mybusybox","tags":["1.1","1.0"]}
```

Pull from the registry

- To pull an image from a registry server you need to know
  - Server URL and port
  - Image repository
  - Image tag

Pull image from registry

```
docker pull myserver.net:5000/my-app:1.0
```

EX5.3 – Pull from the registry

EXERCISE

1. Pair up with another student. Grab their AWS server URL and image repository details
2. On your terminal, run a command to display the tags for the busybox image on your partner’s registry server
   ```
   curl http://<server url>:5000/v2/<name>/mybusybox/tags/list
   ```
3. Try to pull the image with tag 1.0
   ```
   docker pull <server url>:5000/<name>/mybusybox:1.0
   ```
4. What do you notice?

```
$ docker pull 192.241.228.93:5000/johnnytu/mybusybox:1.0
FATA[0000] Error response from daemon: v1 ping attempt failed with error: Get https://192.241.228.93:5000/v1/_ping: tls:
oveived with length 20527. If this private registry supports only HTTP or HTTPS with an unknown CA certificate,
please add `- 192.241.228.93:5000` to the daemon's arguments. In the case of HTTPS, if you have access to the registry's
CA certificate,lag; simply place the CA certificate at /etc/docker/certs.d/192.241.228.93:5000/ca.crt
```

Secure vs insecure registry

- **Secure registry**
  - Has TLS enabled
  - A copy of the CA certificate is placed on the docker host at `/etc/docker/certs.d/<registry url>:5000/ca.crt`
- **Insecure registry**
  - Registry server is not using TLS (just plain HTTP)
  - Registry server is using TLS but the Docker daemon connecting to it does not have the CA certificate
    - Or has the wrong CA certificate

Communicating to an insecure registry

- By default, the Docker daemon assumes all registries are secure and will block communication with insecure registries
  - Cannot `push` or `pull`
- To allow communication with insecure registry, the Daemon must be started with the `--insecure-registry` option and each registry must be added
- For example
  `--insecure-registry myregistry:5000`
- Local registries (IP range in 127.0.0.0/8) are automatically assumed to be insecure and do not need to be added

EX5.4 – Configure insecure registry

EXERCISE

1. Open your `/etc/default/docker` file
2. Add the `--insecure-registry` option into `DOCKER_OPTS` and specify the IP address or domain of your partner's register server
   `DOCKER_OPTS="--insecure-registry <ip address>:5000"`
3. Restart the Docker service
4. Now try and pull the busybox image from your partner's registry again. This time, it should work.
   `docker pull <server url>:5000/<name>/mybusybox:1.0`
5. Check that the image has been pulled into your local box
   `docker images`

Production deployment

- The standard registry image is suitable for evaluation purposes
- For production deployment it is recommended that you build you own registry image
- Building your own image gives you more flexible configuration options
  - Choose storage backend
  - Connect to custom authenticator
  - Enable TLS and create your certificates
- Download the registry 2.0 source from https://github.com/docker/distribution
- For more details on configuring and building see https://docs.docker.com/registry/deploying/#understand-production-deployment

Module Summary

- Your registry server can be publicly available or behind the firewall
- Registries can be secure or insecure
- The official registry image is effective for testing and evaluation
- Production deployments should be run from a custom built registry image

Module 6: Docker Trusted Registry

Module objectives

In this module we will:

- Learn about the 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
  - Pluggable storage drivers
  - Web based GUI for admin configuration
  - Easy and transparent upgrades
  - Built in system usage metrics dashboard
  - Logging

DTR Primary Usage Scenarios

CI/CD with Docker	Containers as a Service
• Centrally located base images • Store individual build images • Pull tested images to production • Developer workflow is a commit	• 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 features

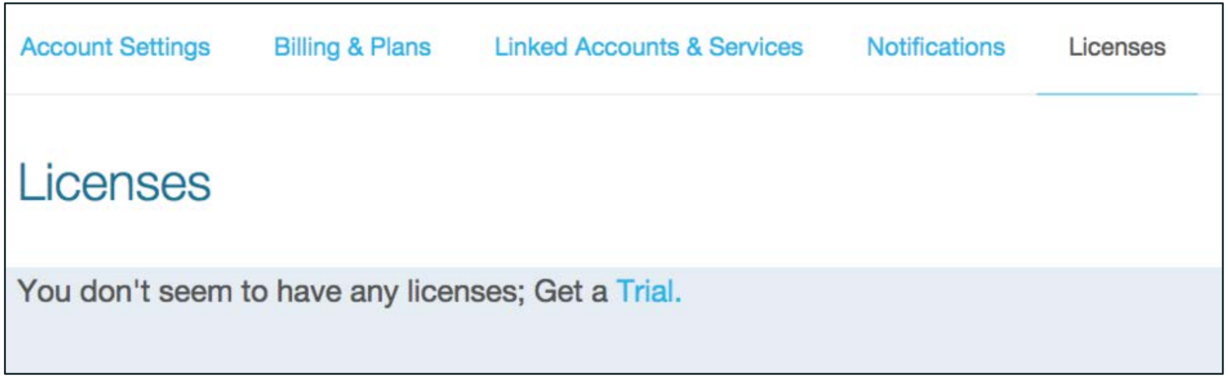
General Features	• Accounts & repos groups UI
CI/CD with Docker	• Image garbage collection • Visual API runner & docs
Registry as a Marketplace	• Image deletion from index • Search & browse index & UI
Platform Features	• Image Provenance • Docker Notary Support

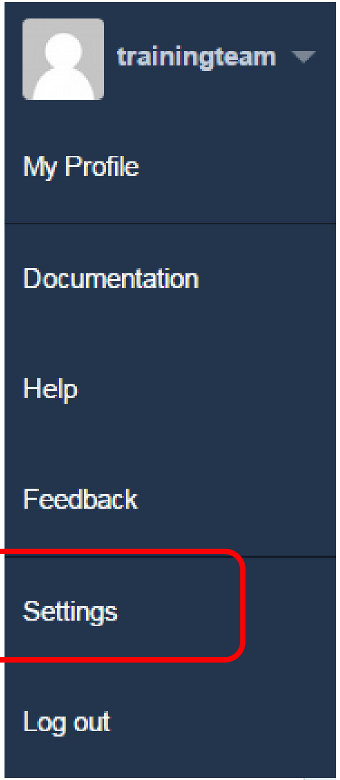
Installation requirements

- DTR is part of Docker Datacenter (DDC) (https://www.docker.com/products/docker-datacenter)
- Requires Datacenter license
- Commercially supported version of Docker Engine (CS Engine)
- CS Engine only supported on RHEL v7.0, 7.1 or Ubuntu 14.04 LTS
- Download the RPM or DEB package from your Enterprise License page in Docker Hub and then copy it into your host machine
- Universal Control Plane (UCP) needs to be installed first
- DTR runs on UCP

Getting access to Docker Datacenter

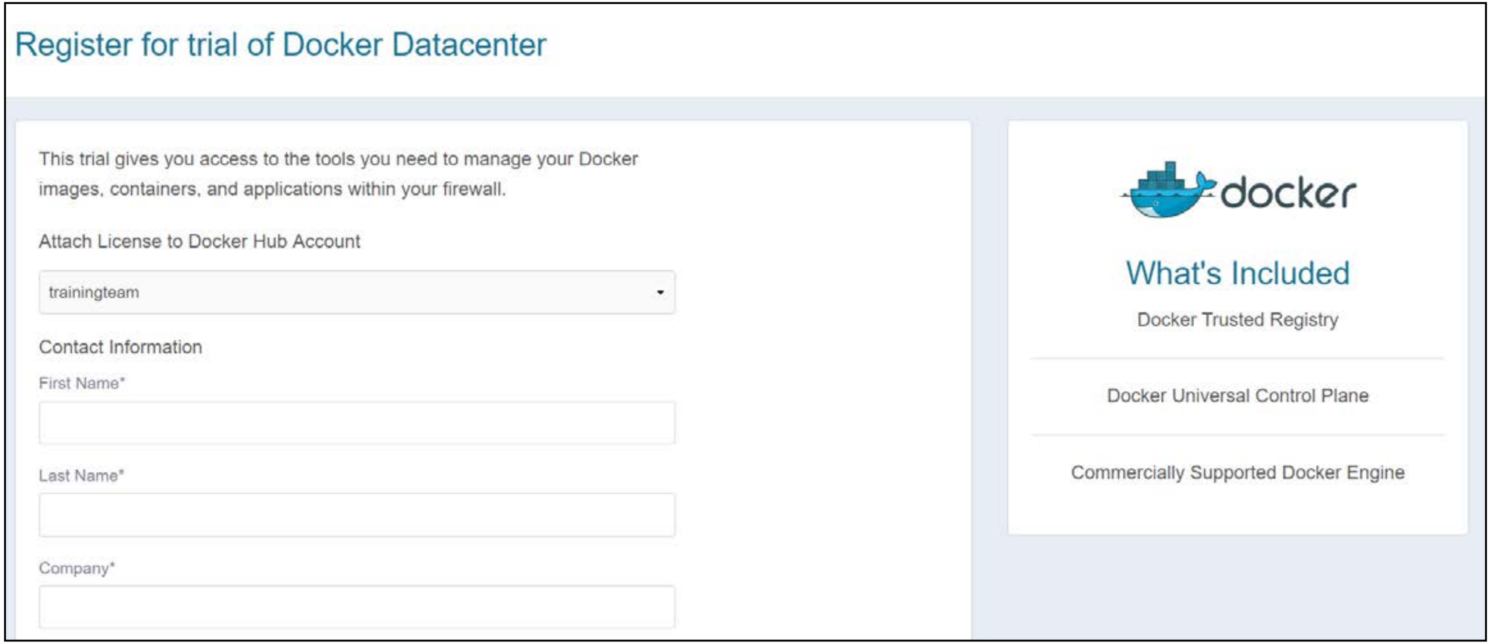
- You can sign up for a 30 day free trial of Docker Datacenter
- Go to your Docker Hub account settings
- Click on the link to “Get a Trial” and complete the registration form





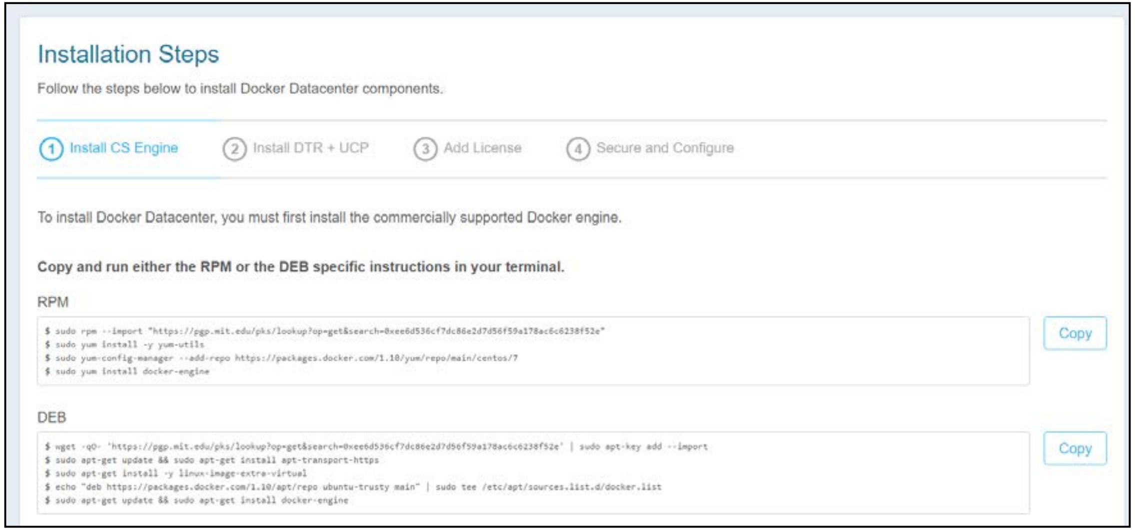
Registration form

- https://hub.docker.com/enterprise/trial/



Installing DTR

- Once you submit the form, you will get your trial license immediately.
- Instructions will be presented to guide you on
  - Installing the CS Engine
  - Installing UCP and DTR
  - Downloading and adding your license



DTR containers

- After you have installed DTR, you should see a number of different containers running
- Each container runs a different function of the application

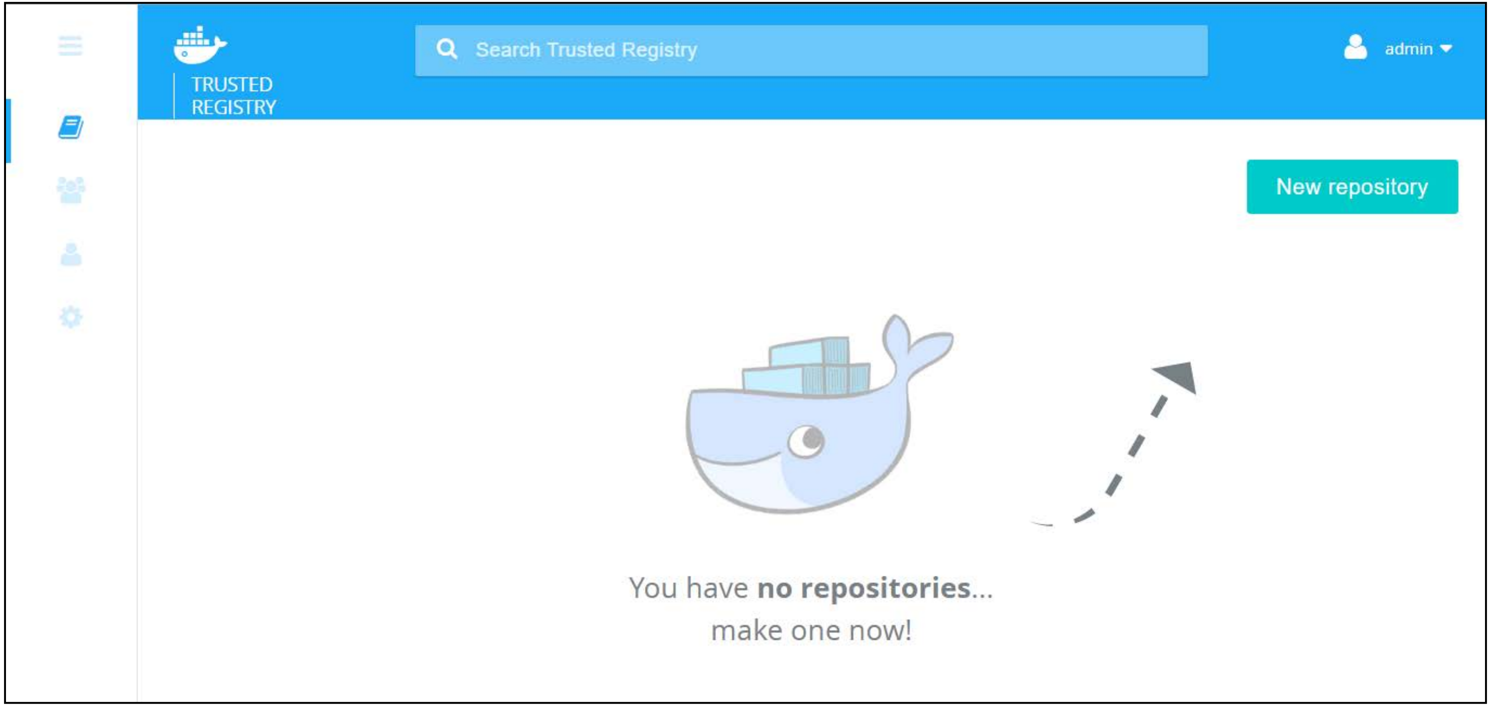
Docker Trusted Registry 2.0.1 - (Replica 419c1c17674a)
RUNNING 5 EXITED 0 Hide Containers (5)

ID	NODE	CONTAINER NAME	IMAGE	CREATED
ab81f78bc5a9	ucp-node-0	dtr-nginx-419c1c17674a	docker/dtr-nginx:2.0.1	2016-06-07 01:13:39 +1000
77f696d0f8b6	ucp-node-0	dtr-api-419c1c17674a	docker/dtr-api:2.0.1	2016-06-07 01:13:25 +1000
42337c1b8143	ucp-node-0	dtr-registry-419c1c17674a	docker/dtr-registry:2.0.1	2016-06-07 01:13:13 +1000
6761e6917423	ucp-node-0	dtr-rethinkdb-419c1c17674a	docker/dtr-rethink:2.0.1	2016-06-07 01:12:58 +1000
6467fe502ba3	ucp-node-0	dtr-etcd-419c1c17674a	sha256:205bc764d14b5c6980ef5812bc0587c30aa22e8c233280d36701e2f347bf4043	2016-06-07 01:12:50 +1000

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

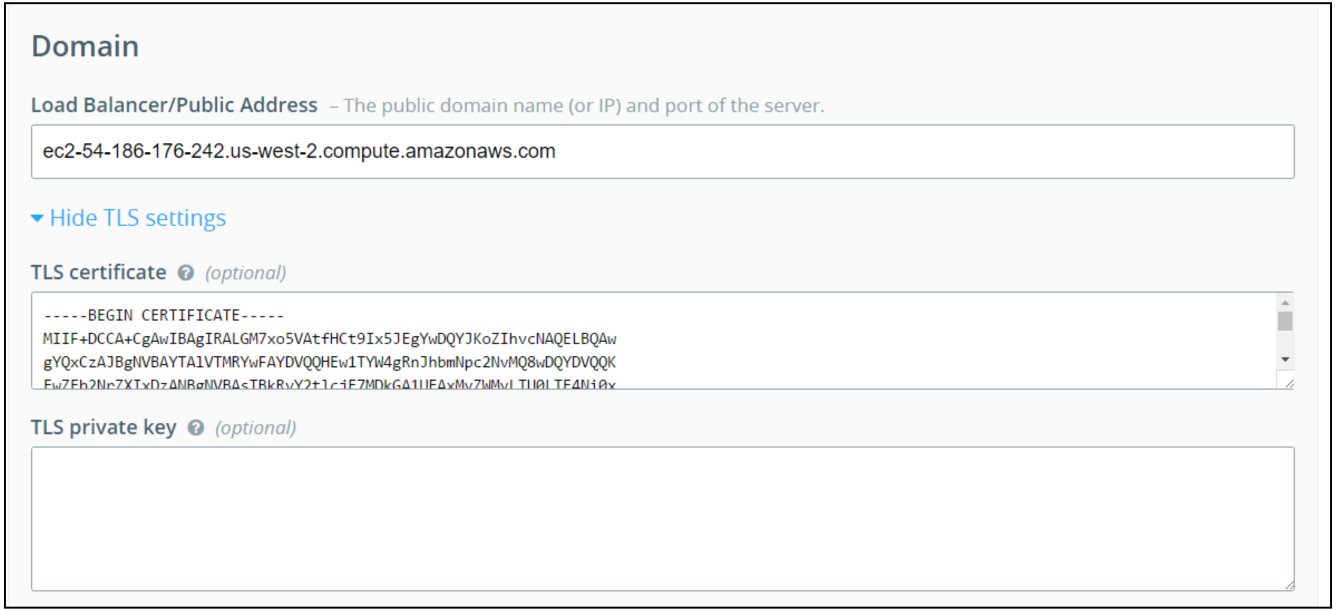


Configuring SSL certificates

- 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-trusted-registry/configure/config-security/#install-registry-certificates-on-client-docker-daemons

Adding your own certificate

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



Insecure Registry

- If the DTR server is not using a trusted certificate AND you have not added the certificate to the Docker daemon, push and pull operations will not be permitted
- Two options
  - Install the certificates
  - Add the `--insecure-registry` flag to the Docker daemon startup flags
- With the `--insecure-registry option,` communication is still secure but the Docker daemon is not confirming that the Registry connection is not being hijacked or diverted.

Untrusted certificates

- When attempting push and pull operations or logging into DTR on the CLI you will get the following error if the DTR certificates are not trusted

```
ubuntu@node-0:~$ docker login ec2-54-186-176-242.us-west-2.compute.amazonaws.com
Username: admin
Password:
Error response from daemon: Get https://ec2-54-186-176-242.us-west-2.compute.amazonaws.com/v1/users/:
x509: certificate signed by unknown authority
```

Installing DTR certificates on client daemons

- Certificates need to be installed on each Docker daemon that needs to connect to DTR

```
$ export DOMAIN_NAME=dtr.yourdomain.com
```

Get certificate from DTR and install

```
$ openssl s_client -connect $DOMAIN_NAME:443
-showcerts </dev/null 2>/dev/null | openssl x509
-outform PEM | sudo tee /usr/local/share/ca-
certificates/$DOMAIN_NAME.crt
```

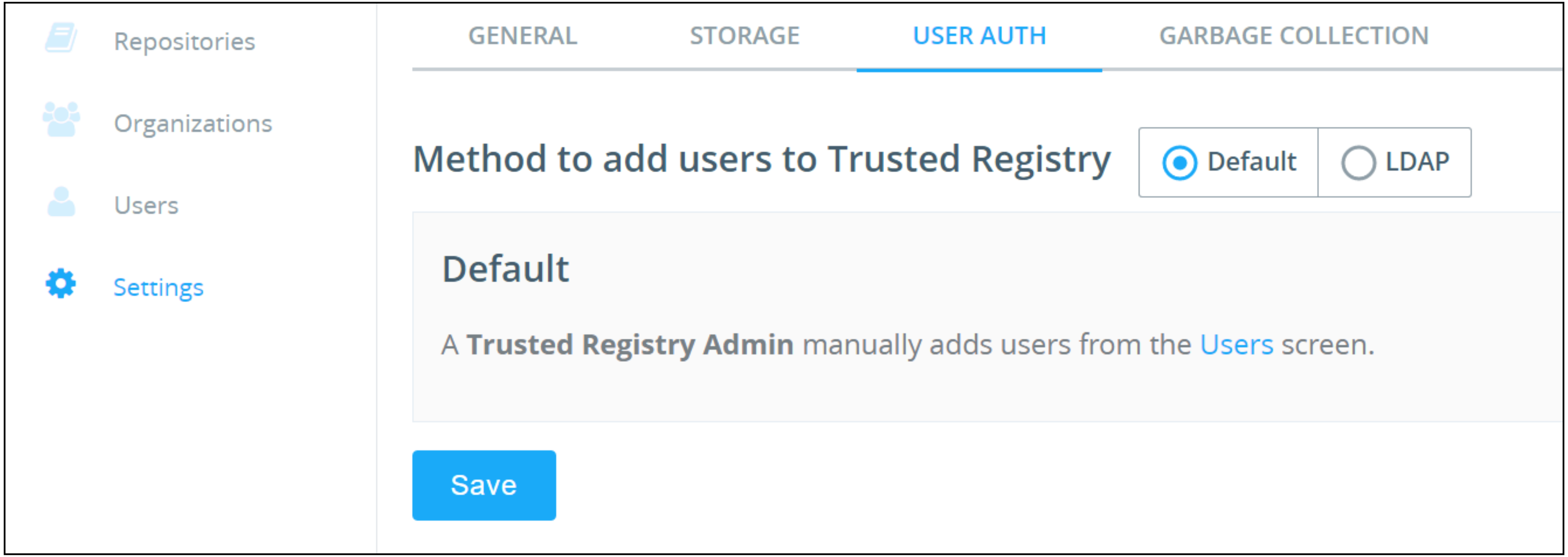
Update certificates and then restart Docker

```
$ sudo update-ca-certificates
$ sudo service docker restart
```

User authentication

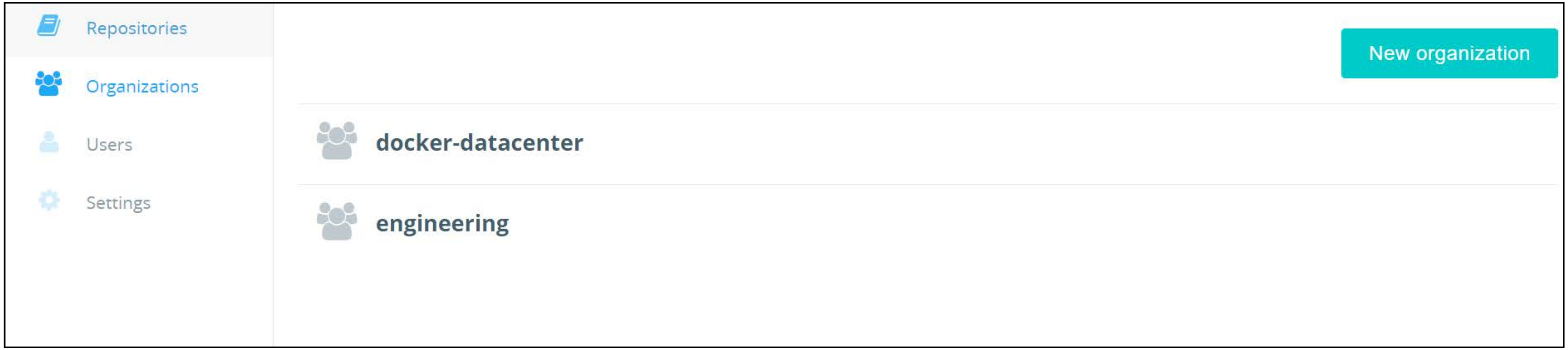
- There are two options for authentication
  - Default
  - LDAP
- Default authentication uses a simple username and password list to control who has access and what level of access
- LDAP authentication connects DTR to your LDAP server and uses the users on that server

Choosing user authentication method



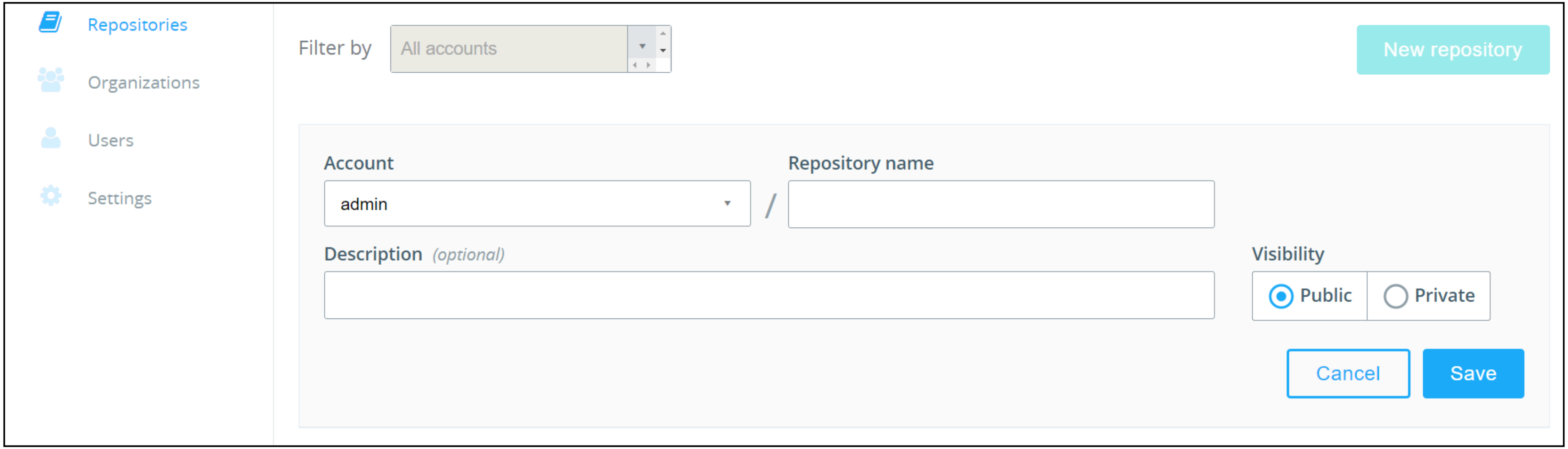
Organizations and Teams

- Allow us to group users and control the level of access to repositories
- Organizations can consists of multiple teams
- Users are assigned to teams



Creating a repository

- Repositories can be associated with a user account, an organization or a team



Pushing images to DTR

- Same process as pushing an image to any registry server
- Tag the image with the DTR server URL and then push
- In order to push images to DTR, the repository must be created first
- Authentication must be enabled and you will need to login by using the `docker login` command (more on this later)

Tag an image with the DTR server url

```
docker tag <image id> <DTR server url>/jtu/myapp:1.0
```

Push the image to DTR

```
docker push <DTR server url>/jtu/myapp:1.0
```

Unauthorized action

- With authentication enabled, users cannot push and pull images unless they login to their user account on DTR
- Without a login, users will see the following message when trying to push an image

```
student@masterhost:~$ docker push 107.170.251.32/admin/mybusybox:1.0
The push refers to a repository [107.170.251.32/admin/mybusybox] (len: 1)
c51f86c28340: Preparing
unauthorized: authentication required
```

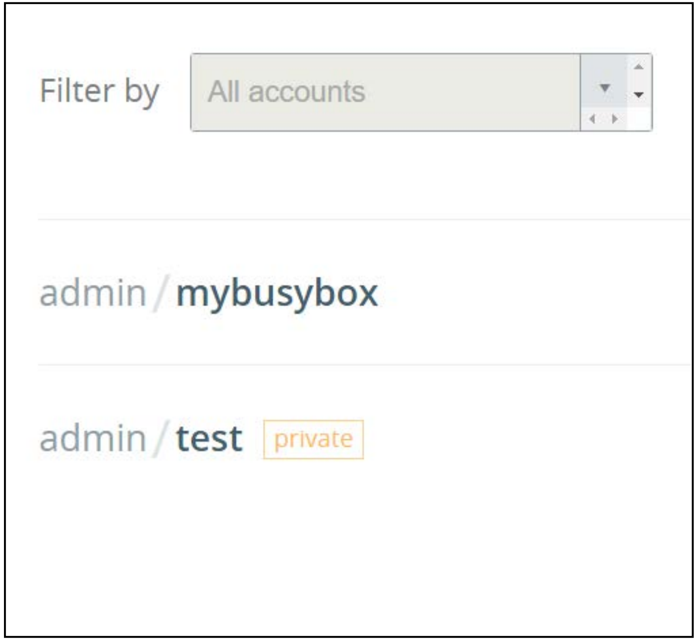
Login to DTR on the Docker client

- Use the `docker login` command and specify the DTR server URL
- You will be prompted for a username and password
- Command
  `docker login <server url>`
- To logout use
  `docker logout <server url>`

```
$ docker login 107.170.229.60
Username: admin
Password:
Email:
WARNING: login credentials saved in /home/johnnytu/.dockercfg.
Login Succeeded
johnnytu@docker-ubuntu:~$
```

Public vs private repositories

- Public repositories created under a user account are openly available
  - Can pull images without authentication, even when managed authentication is enabled
  - Still need to login to push images
- Private repositories created under a user account can only be accessed by:
  - The user who created the repo
  - Any user with a global role



Pulling images from DTR

- Same process as pulling images from any other registry server or from Docker Hub
- Specify the DTR server URL in the image tag

Pull the jtu/myapp:1.0 image

```
docker pull <DTR server url>/jtu/myapp:1.0
```

Run a new container using the jtu/myapp:1.0 image. Remember that the run command will pull the image first if it doesn't exist on the host

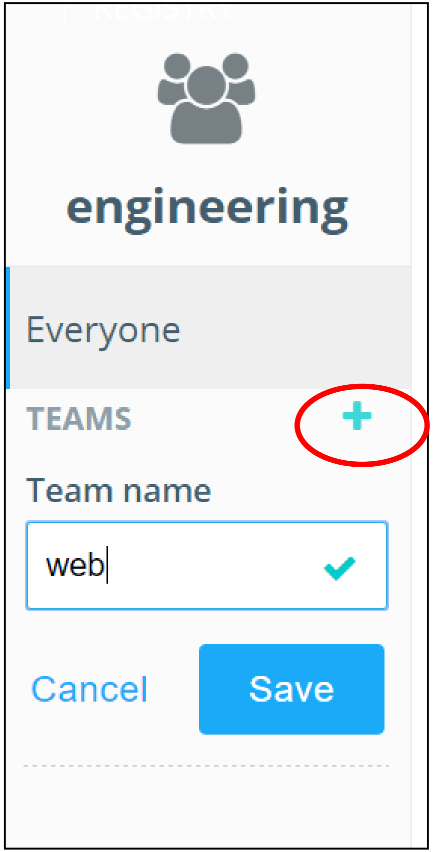
```
docker run <DTR server url>/jtu/myapp:1.0
```

Creating an Organization

- Give the organization a name
- Name will be used in organization repositories

Creating teams

- Organizations can be organised into multiple teams
- Teams are configured with individual users as members

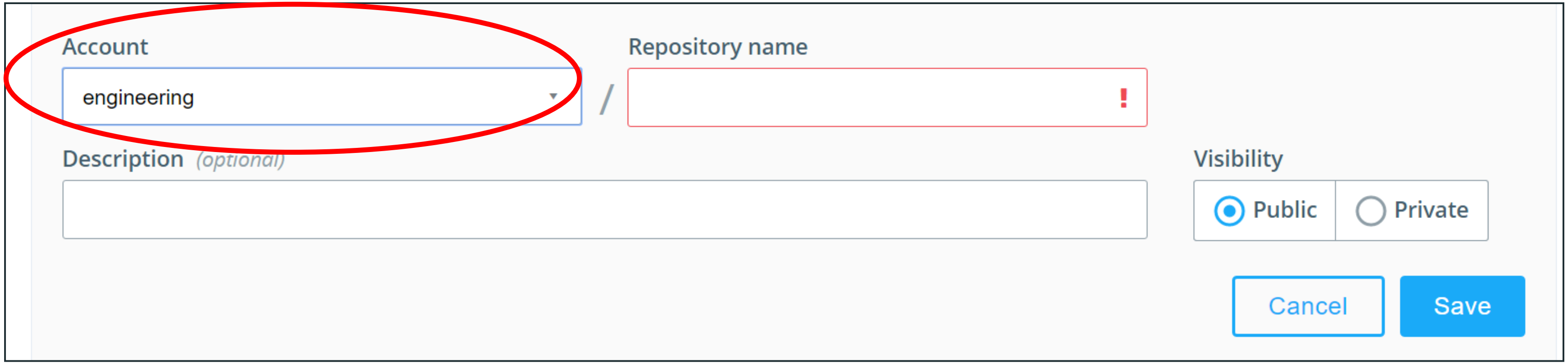


Organization and team members

- Can add users directly to an organization or in one or more of the organization teams
- Team members are automatically organization members
- Organization members can
  - View private repositories in the organization
  - View teams and other members in the organization
- Team members can
  - Access private team repositories
- **Note:** Admin users can access all repositories in all organizations and teams
- **Note:** Public organization and team repositories are visible to everyone

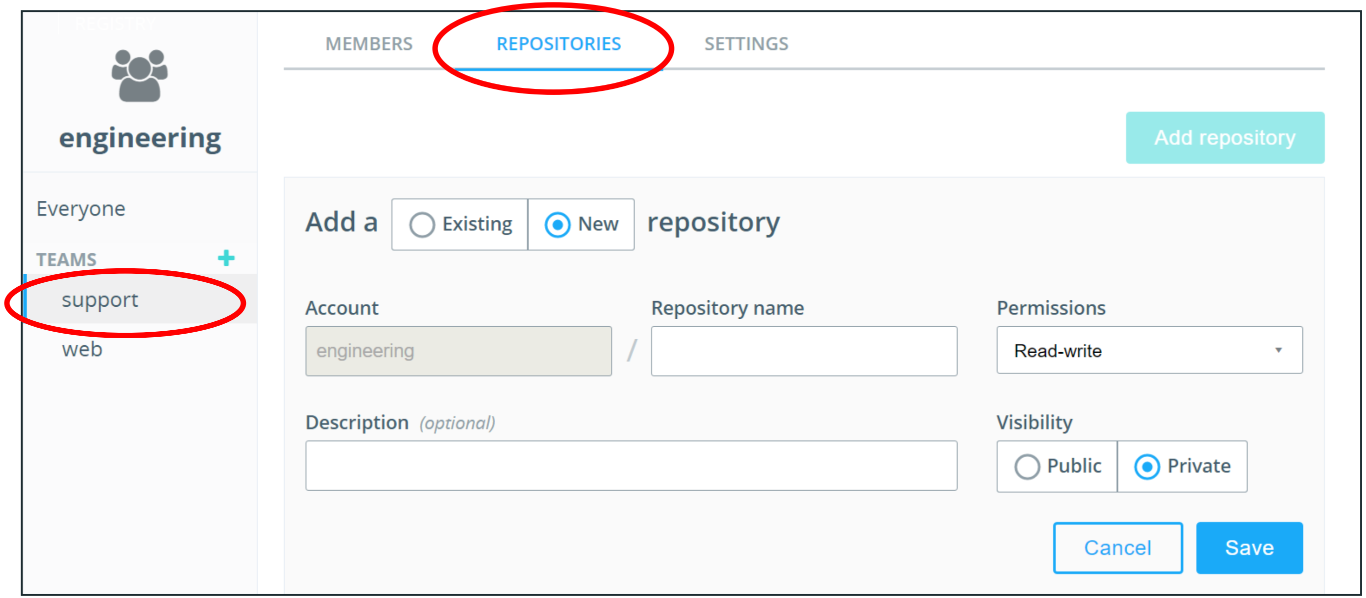
Creating organization repos

- To create organization repositories, you must be a member of the organization
- Repositories are prefixed with the organization name (i.e `my-organization/my-repo:1.0`)
- Repositories can be public or private



Creating team repos

- To create team repositories, you must be a organization admin
- Repo’s can belong in multiple teams

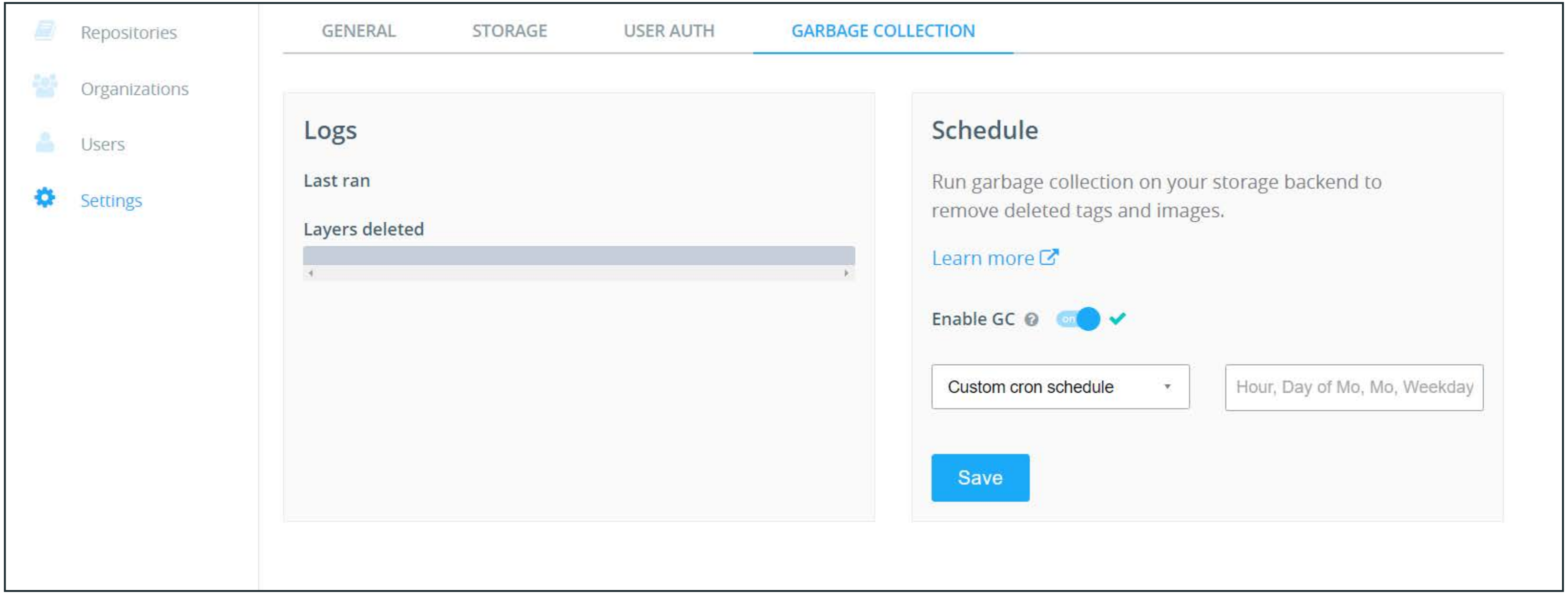


Team repository permissions

- In addition to public and private repositories, team repositories can have a set permission
  - Admin
  - Read-write
  - Read only
- Permission controls the access level of team members
- Example setup
  - Allow “dev” team to have read only access to a production image repo
  - Allow “ops” team to have read-write access

Garbage Collection and Image Deletion

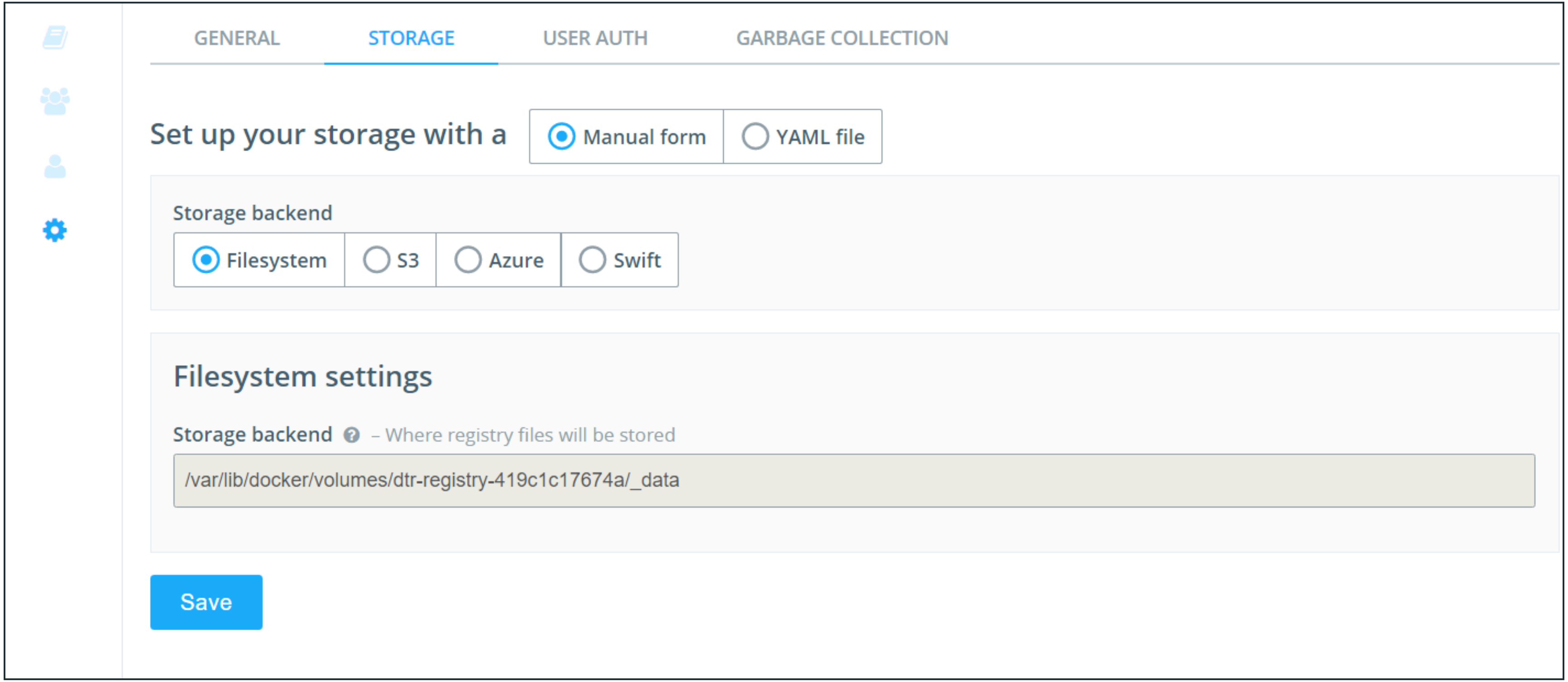
- Soft Image Delete = Remove from UI but still on disk
- Hard Image Delete = Remove unused layers via garbage collection



LDAP authentication

- Connect to LDAP server for authentication
- Usernames and passwords will be based on what is configured on the LDAP server
- Your LDAP server may have thousands of users
  - Do all of them need access?
- Filter for DTR registry users and DTR administrators

Storage Backends



Module summary

- DTR is part of Docker Datacenter and is installed from the Universal Control Plane (UCP)
- UCP 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
- Users can be organized into organizations and teams
- Repositories can belong to individual users, organizations or teams
- Repositories can be public or private

Module 7: Docker Machine

Module objectives

In this module we will:

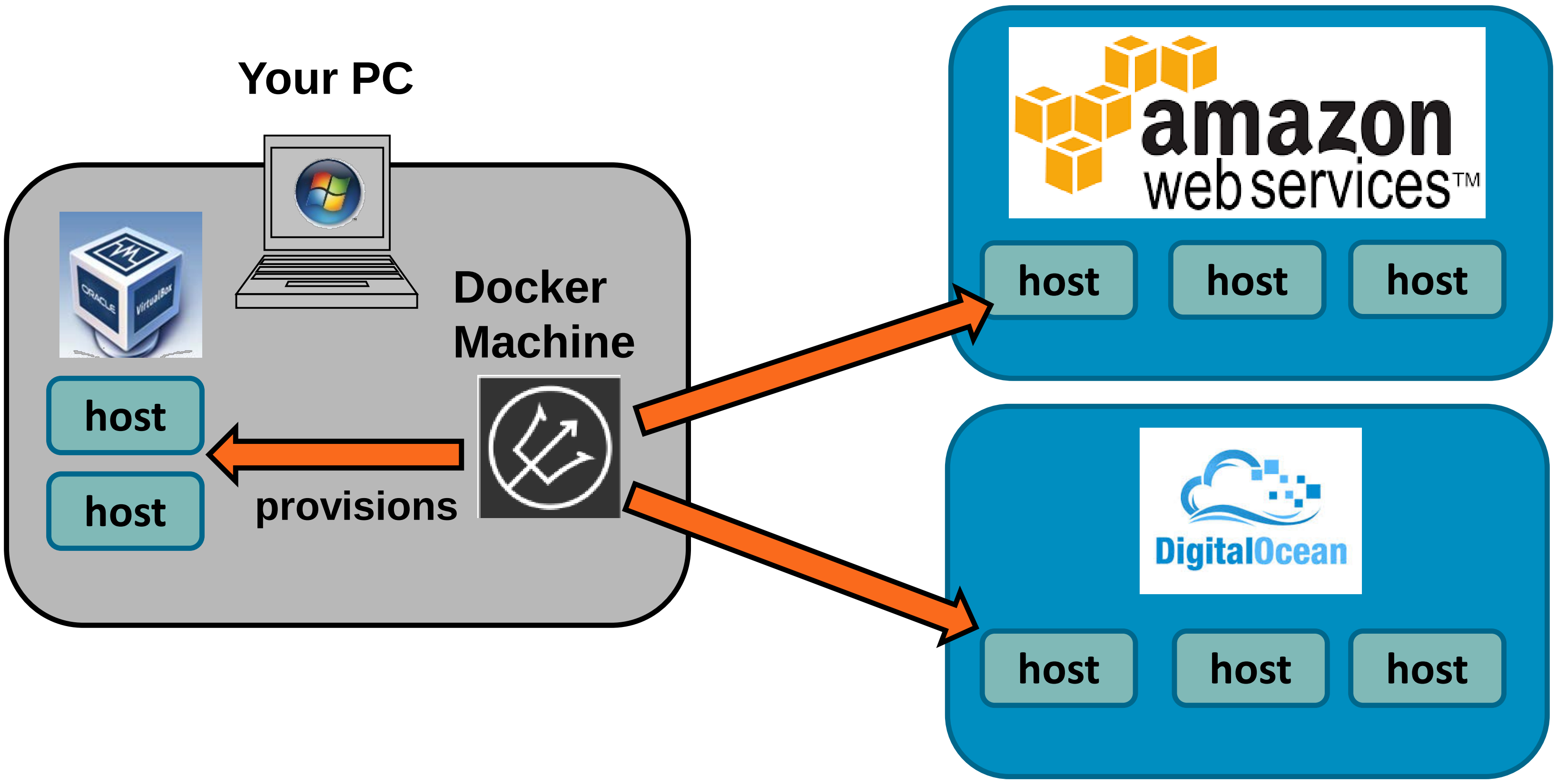
- Learn how to install Docker Machine on Linux, Windows and OSX
- Use Docker Machine to provision Docker hosts on a local virtualization platform and in the cloud
- Learn the key commands to manage our hosts that have been provisioned by Docker Machine

Docker Machine overview

***Docker Machine** is a tool that automatically provisions Docker hosts and installs the Docker Engine on them*

- Create additional hosts on your own computer
- Create hosts on cloud providers (e.g. Amazon AWS, DigitalOcean etc…)
- Machine creates the server, installs Docker and configures the Docker client

Docker Machine overview



Installing Machine

- Download the binary zip release for the operating system at https://github.com/docker/machine/releases
- Zip folder contains the Docker Machine binary and the driver binaries
- Unzip and place the binaries into a folder of your choice
- Add the folder to your system environment PATH
- Recommended to place into a folder that is on your system environment path
  - (i.e. /usr/local/bin)

Installing machine on Linux

- Download the Linux zip from https://github.com/docker/machine/releases/download/v0.5.2/docker-machine_linux-amd64.zip
- Unzip and move all the binaries into `/usr/local/bin`

Verify installation

- Run `docker-machine -v` to display the version number
- Run `docker-machine` and you should see a list of appropriate commands

EX7.1 – Install machine on Ubuntu

EXERCISE

Use your master instance

1. Download the Linux binary from https://github.com/docker/machine/releases/download/v0.5.2/docker-machine_linux-amd64.zip
2. Unzip the file into the `/usr/local/bin` folder (you may need to install the unzip program)

```
$ sudo apt-get install unzip
$ unzip docker-machine_linux-amd64.zip -d
/usr/local/bin
```

Install Machine on Windows and OSX

- Best option is to use Docker Toolbox
- Toolbox will install Docker Machine and the Docker client into your environment
- To install manually, download the appropriate Docker Machine zip from https://github.com/docker/machine/releases
  - Unzip and place all the binaries into your environment path
  - Windows users should use `Msysgit` as their terminal instead of `CMD` if installing Docker Machine manually

EX7.2 – Install Docker Toolbox

EXERCISE

1. Go to https://www.docker.com/docker-toolbox. Download and install Docker Toolbox on your laptop. This will install Docker Machine as well

For Mac OSX Users

2. Open your terminal and run `docker-machine -v` and make sure you can see the Docker Machine version

For Windows Users

2. Open your CMD command line terminal and run `docker-machine -v` and make sure you can see the Docker Machine version

Using docker machine

- The `docker-machine` binary has commands to create and manage Docker hosts in a variety of environments such as:
  - VirtualBox
  - Amazon AWS
  - DigitalOcean
  - Azure
  - Rackspace
  - And more …
- We will looks at examples for VirtualBox, AWS and DigitalOcean
- Each environment has its own plugin binary, which is distributed in the zip file download

Creating a host

- Use **`docker-machine create`** command and specify the driver to use
- The driver allows docker-machine to interact with the environment where you want to create the host
- Syntax
  `docker-machine create --driver <driver> <hostname>`

Creating a host on VirtualBox

- Using VirtualBox allows us to quickly provision additional Docker hosts on our Windows or Mac
- Must have VirtualBox installed
  https://www.virtualbox.org/wiki/Downloads
- Use `virtualbox` driver

Create a host named "testhost" on the current machine, using Virtual Box.

```
docker-machine create --driver virtualbox testhost
```

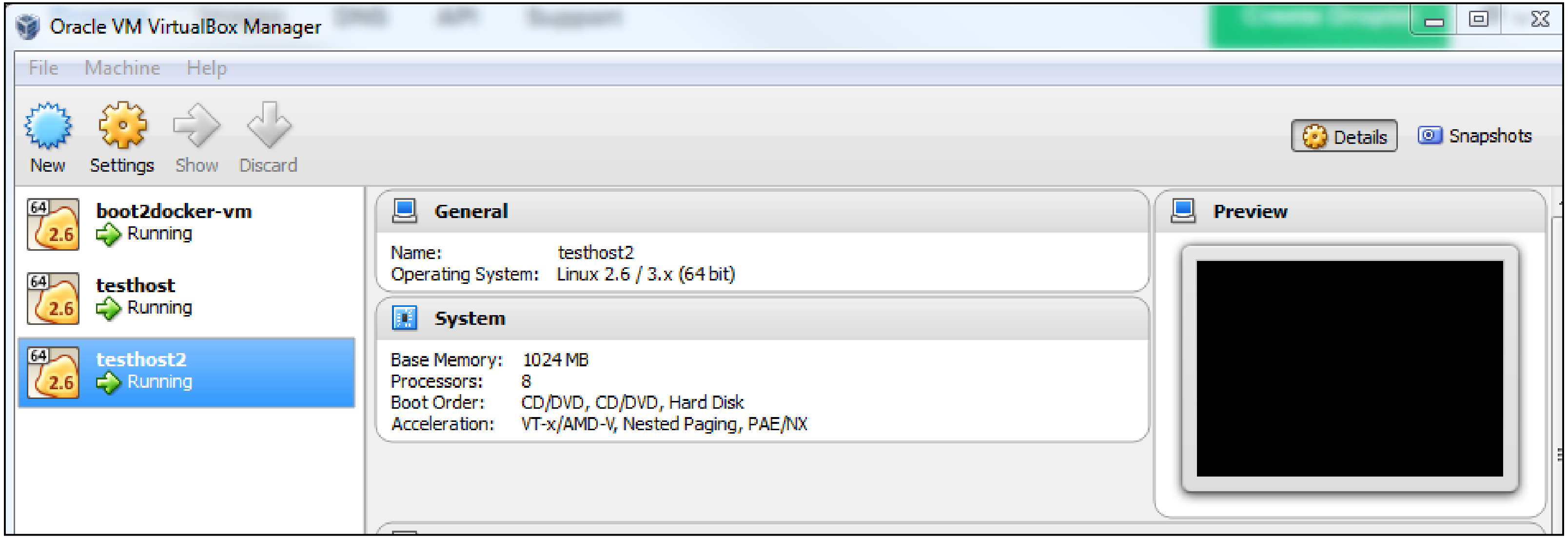
Creating a host on VirtualBox

- `docker-machine` will download the boot2docker Linux distribution, create and start a VirtualBox VM which has Docker running on it
- `docker-machine` will automatically create the SSH key for your host

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine create --driver virtualbox testhost2
←[34mINFO←[0m[0000] Creating SSH key...
←[34mINFO←[0m[0000] Creating VirtualBox VM...
←[34mINFO←[0m[0021] Starting VirtualBox VM...
←[34mINFO←[0m[0025] Waiting for VM to start...
←[34mINFO←[0m[0093] "testhost2" has been created and is now the active machine.
←[34mINFO←[0m[0093] To point your Docker client at it, run this in your shell:
eval "$(C:\Program Files (x86)\Git\bin\docker-machine env testhost2)"
```

Check your hosts on VirtualBox

- You can check to ensure that your hosts are in VirtualBox



EX7.3 – Create machine on VirtualBox

EXERCISE

1. Open your PC or Mac terminal. For PC users, use msysgit
2. Create two hosts on VirtualBox called testhost1 and testhost2
```
docker-machine create --driver virtualbox testhost1
docker-machine create --driver virtualbox testhost2
```
3. Open VirtualBox and check that both hosts are present

Provisioning hosts in the cloud

- Each cloud provider has different options on the **docker-machine create** command and their own driver
- Full list of drivers
  - Amazon Web Services
  - Google Compute Engine
  - IBM Softlayer
  - Microsoft Azure
  - Microsoft Hyper-V
  - Openstack
  - Rackspace
  - Oracle VirtualBox
  - VMware Fusion
  - VMware vCloud Air
  - VMware vSphere

Creating hosts in DigitalOcean

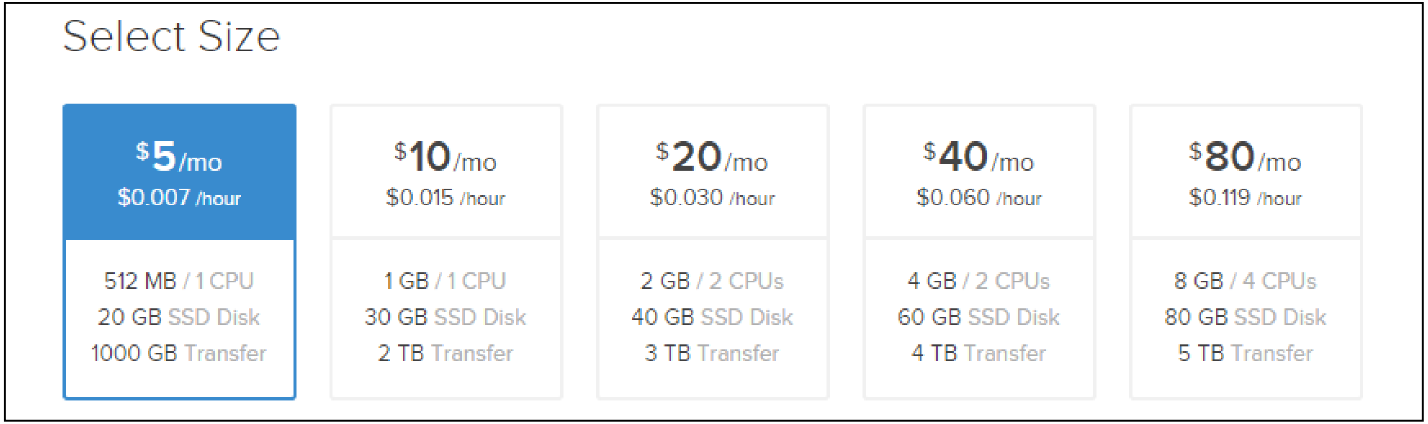
- You will need your DigitalOcean account access token
- Default droplet size is 512mb
- Default image is Ubuntu 14.04 (`ubuntu-14-04-x64`)
- Default region is `nyc3`

Example with DigitalOcean

```
docker-machine create
      --driver digitalocean \
      --digitalocean-access-token <your access token> \
      --digitalocean-size 2gb \
      testhost
```

DigitalOcean droplet size

- Use the `--digitalocean-size` option to specify.
- Size amounts correspond to the amount of RAM allocated to a droplet



Output

```
$ docker-machine create --driver digitalocean \
                        --digitalocean-access-token <access token here> \
                        machine-host1
INFO[0000] Creating CA: /home/johnnytu/.docker/machine/certs/ca.pem
INFO[0000] Creating client certificate: /home/johnnytu/.docker/machine/certs/cert.pem
INFO[0001] Creating SSH key...
INFO[0002] Creating Digital Ocean droplet...
INFO[0180] "machine-host1" has been created and is now the active machine.
INFO[0180] To point your Docker client at it, run this in your shell: eval "$(docker-machine env machine-host1)"
```

Creating hosts in AWS

- To create hosts in AWS, you will need your
  - AWS access key for the API
  - AWS secret key for the API
  - The VPC ID to launch the instance in
- Default image used is Ubuntu 14.04 LTS

```
docker-machine create
      --driver amazonec2 \
      --amazonec2-access-key <AWS access key> \
      --amazonec2-secret-key <AWS secret key> \
      --amazonec2-vpc-id <VPC ID> \
      testhost
```

Creating hosts in AWS

- Some other options you can specify include:
  - Amazon Machine Image ID (`--amazonec2-ami`)
  - Instance type (`--amazonec2-instance-type`)
    - Default is `t2.micro`
  - Which region to use (`--amazonec2-region`)
    - Default is `us-east-1`
  - Root disk size (`--amazonec2-root-size`)
    - Default is `16GB`

- Full list of options at https://docs.docker.com/machine/#amazon-web-services

The underlying process

- First, machine will create a SSH key that is to be used to provision the host and also to access the host
- The SSH key is stored in `/home/<user>/.docker/machines` for Linux and OSX and `C:\Users\<user>\.docker\machine` on Windows
- Machine will then install Docker on the host and configure the Docker daemon to accept remote connections over TCP
- TLS will be enabled for authentication
- Server certificate and key stored in `/etc/docker` folder on remote host
- Client certificate and key stored in same folder as ssh key

EX7.4 – Provision cloud hosts

EXERCISE

1. On your Ubuntu AWS master instance, use `docker-machine` to create two hosts in a Cloud environment.
   Your instructor will provide you with the necessary details
2. Alternatively, if you have your own AWS or DigitalOcean account or an account with any other supported provider you may create the hosts there
3. Call your hosts `cloudhost1` and `cloudhost2` and prepend it with your name. For example:
   `jtucloudhost1`
4. Repeat the same steps but this time, provision the host from the `docker-machine` running on your PC or Mac. Call your host `cloudhostfrompc` and prepend with your name
   `jtucloudhostfrompc`

Finding your hosts

- If you created hosts in VirtualBox, just open VirtualBox and you will see the hosts there.
- For cloud hosts, you can login to your cloud provider account and view the instances / droplets that have been created
- But what if have created hosts in many different providers such as AWS, DigitalOcean, Rackspace, Azure?
- We don’t want to have to login to each account to find out host machines

List your machines

- The `docker-machine ls` command displays all the host machines that have been provisioned
- Can easily see hosts across different cloud providers

```
Johnny@JTCOMMANDCENTER ~
$ docker-machine ls
NAME          ACTIVE   DRIVER         STATE     URL                         SWARM
cloudhost1    *        digitalocean   Running   tcp://45.55.213.23:2376
testhost               virtualbox     Running   tcp://192.168.99.100:2376
testhost2              virtualbox     Running   tcp://192.168.99.101:2376
```

EX7.5 – List machines

EXERCISE

1. Using your Ubuntu AWS master instance, list out all the hosts provisioned by docker-machine
   `docker-machine ls`
2. Go to your home directory and run `ls –a`
3. Confirm that you can see the hidden `.docker` folder
4. Go into the `.docker` folder and inside, check for the `machine` folder and change directory into machine.
5. Then check for the `machines` folder and change directory into it
6. What do you notice in the machines folder
   `(/home/<user>/.docker/machine/machines)`

Connecting to a host machine

- We can’t just use regular SSH to get access to our host because we don’t know the username and password
- There are 2 methods to connect to a host that docker-machine has provisioned
  - Use docker-machine ssh
  - Set the environment variables to point your Docker client to the daemon on the remote host

docker-machine env command

- The env command prints out the environments variables that need to be set in order to connect your Docker client, to the remote daemon of the specified host
- Syntax
  `docker-machine env <hostname>`

```
$ docker-machine env machine-host1
export DOCKER_TLS_VERIFY=1
export DOCKER_CERT_PATH="/home/johnnytu/.docker/machine/machines/machine-host1"
export DOCKER_HOST=tcp://104.236.121.222:2376
```

Using environment variables

- Run `eval $(docker-machine env <hostname>)` to point your Docker client to the daemon on the host specified
  - Works by setting environment variables on the client
  - Much easier than manually setting the variables one at a time
- **For windows users:** This will only work if you are using the `msysgit` terminal

Connects local docker client to docker daemon on host3

```
eval $(docker-machine env host3)
```

Disconnects the docker client from the daemon on host3 by resetting the environment variables

```
eval $(docker-machine env –u)
```

Checking the Active Host

- If you run `docker-machine ls` you will notice a column called "`ACTIVE`", marking the machine which is the active host
- The active host is the machine that the Docker client is connected to
- Active host is set when running the `env` command
  `eval $(docker-machine env <hostname>)`
- Can run `docker-machine active` to print the name of the active host

Setting the active host

```
student@DockerTraining:~$ docker-machine ls
NAME          ACTIVE   DRIVER         STATE     URL                           SWARM
cloudhost              digitalocean   Running   tcp://159.203.84.124:2376
cloudhost2             digitalocean   Running   tcp://159.203.83.120:2376
student@DockerTraining:~$ eval $(docker-machine env cloudhost)
student@DockerTraining:~$ docker-machine ls
NAME          ACTIVE   DRIVER         STATE     URL                           SWARM
cloudhost     *        digitalocean   Running   tcp://159.203.84.124:2376
cloudhost2             digitalocean   Running   tcp://159.203.83.120:2376
student@DockerTraining:~$
```

EX7.6 – Connect client to remote host

EXERCISE

1. Using your Ubuntu AWS master instance, connect your Docker client to cloudhost1
   `eval $(docker-machine env cloudhost1)`
2. Check your active host to make sure the Docker client is connected to `cloudhost1`
   `docker-machine active`
3. Run a few containers of your choice on the host. Remember what you have run. You will need it later.

Disconnecting the client

- To disconnect the Docker client from the remote daemon we need to unset the variables
  - `DOCKER_TLS_VERIFY`
  - `DOCKER_CERT_PATH`
  - `DOCKER_HOST`
- The `docker-machine env -u` command option prints out the variables you need to unset
- You can use the unset command or as a shortcut run
  `eval $(docker-machine env -u)`

EX7.7 – Disconnect and connect again

EXERCISE

1. Disconnect your Docker client from the remote daemon
   `eval $(docker-machine env -u)`
2. Connect it to the other host you provisioned
   `eval $(docker-machine env <hostname>)`
3. Run `docker ps -a` and notice how there are no previous containers
4. Run a few containers
5. Disconnect the client from the remote daemon. The client should be back on your localhost. Connect back to the first remote host
6. Run `docker ps -a`. Can you see the containers you ran in exercise 7.6?

Docker machine SSH

- The `docker-machine ssh` command allows us to connect to a provisioned host using SSH
- Logs in using the SSH key that is created when creating the machine
- Can also be used to run a command on the specified machine

Connect to host3 using SSH

```
docker-machine ssh host3
```

Running commands with ssh

- You can use the ssh command to run any process that is available on the specified host
- Syntax
  ```
  docker-machine ssh <machine name> <command>
  ```
- If you specify arguments in your command you must put the flag parsing terminator (`--`) before your command

Check processes running on the machine

```
docker-machine ssh <machine name> ps
```

List all files on root folder of the machine

```
docker-machine ssh <machine name> -- ls –l /
```

EX7.8 – SSH into a host (optional)

EXERCISE

1. Using your PC or Mac terminal, connect to one of the VirtualBox VM's you created using SSH
   ```
   docker-machine ssh <host name>
   ```
2. Run a few containers of your choice and then exit the host
3. Use the ssh command to check what containers are running on the host you just exited
   ```
   docker-machine ssh <host name> docker ps
   ```
4. Run docker-machine ssh <host name> ps -ef. Notice the error
5. Fix the command in 4)
   ```
   docker-machine ssh <host name> -- ps -ef
   ```

Start and Stop hosts

- To stop a host machine
  `docker-machine stop <machine name>`
- To start a stopped host machine
  `docker-machine start <machine name>`
- To restart a host machine
  `docker-machine restart <machine name>`

```
$ docker-machine ls
NAME          ACTIVE   DRIVER         STATE     URL                         SWARM
cloudhost1    *        digitalocean   Running   tcp://45.55.213.23:2376
testhost               virtualbox     Running   tcp://192.168.99.100:2376
testhost2              virtualbox     Running   tcp://192.168.99.101:2376

$ docker-machine stop testhost

$ docker-machine ls
NAME          ACTIVE   DRIVER         STATE     URL                         SWARM
cloudhost1    *        digitalocean   Running   tcp://45.55.213.23:2376
testhost               virtualbox     Stopped
testhost2              virtualbox     Running   tcp://192.168.99.101:2376
```

EX7.9 – Start and stop machines

EXERCISE

This exercise is optional

1. Using your PC (`msysgit`) or Mac terminal, stop one of the VirtualBox hosts you created previously
2. Run `docker-machine ls` and verify the machine state
3. Now start the host up
4. Open VirtualBox and manually stop the other host
5. Run `docker-machine ls` and verify the machine state (might take a while for it to get the state information)
6. Now start the host again

Inspecting host details

- Use the `docker-machine inspect` command to get details on the host machine
- Details include driver info, certificate paths, IP address, store path etc…

```
johnnytu@docker-ubuntu:~$ docker-machine inspect
{
    "DriverName": "digitalocean",
    "Driver": {
        "AccessToken": "21eaff99811daac4e4a9cf5d7e6f2a33d6886ad237278ac21847f47985b9d1ac",
        "DropletID": 5423809,
        "DropletName": "",
        "Image": "ubuntu-14-04-x64",
        "MachineName": "machine-host4",
        "IPAddress": "45.55.144.127",
        "Region": "nyc3",
        "SSHKeyID": 813896,
        "SSHUser": "root",
        "SSHPort": 22,
        "Size": "512mb",
        "IPv6": false,
        "Backups": false,
```

Getting the IP address of a host

- You can see the IP address by looking at the URL column on the output of `docker-machine ls`
- However sometimes it's more effective to run `docker-machine ip <host name>` as this will only output the IP
- You can feed the output as an argument for another command

```
$ docker-machine ip testhost
192.168.99.100

$ ping $(docker-machine ip)
Pinging 45.55.213.23 with 32 bytes of data:
Reply from 45.55.213.23: bytes=32 time=280ms TTL=50
Reply from 45.55.213.23: bytes=32 time=328ms TTL=50
Reply from 45.55.213.23: bytes=32 time=324ms TTL=50
```

Deleting hosts

- To delete a host, use `docker-machine rm` command
- This will remove the host on the environment (the virtualization platform or cloud provider) and delete the local reference folder (`/home/<user>/.docker/machine/machines/<machine name>`)
- If you manually delete the host from the cloud provider or VM platform Docker Machine will still store a reference to it locally.
  - Error's will occur when running commands such as `docker-machine` ls as it won't be able to find the host

Example

```
docker-machine rm host1
```

EXERCISE

EX7.10 – Delete hosts

1. Change directory into /home/<user>/.docker/machine/machines
2. List the folder in this directory and notice how each machine you provisioned has its own folder
3. **Using your AWS Ubuntu master host,** delete the two other hosts that you provisioned earlier
   docker-machine rm <host name>
4. List the files in the directory again and verify that the machines have been deleted. Also verify this by running docker-machine ls
5. Open VirtualBox and manually delete one of the hosts
6. Open your local **PC or Mac terminal (use msysgit for PC)** and run docker-machine ls. Notice the error message for the host you manually deleted
7. Delete the local reference to the host machine that is causing the error

Summary

- Docker Machine provides an effective way for us to provision and manage Docker hosts across multiple virtualization and cloud environments
- Hosts that are provisioned by Docker Machine have the Docker daemon ready to accept remote connections over TCP. The daemon is secured by TLS
- Docker Machine can be used to SSH into a host or to execute a command on a host

Module 8:
Docker Swarm

Module objectives

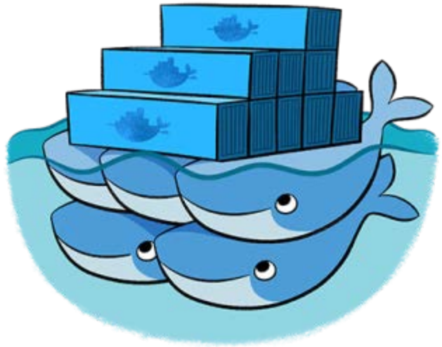
In this module we will:

- 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
- Learn about the Universal Control Plane

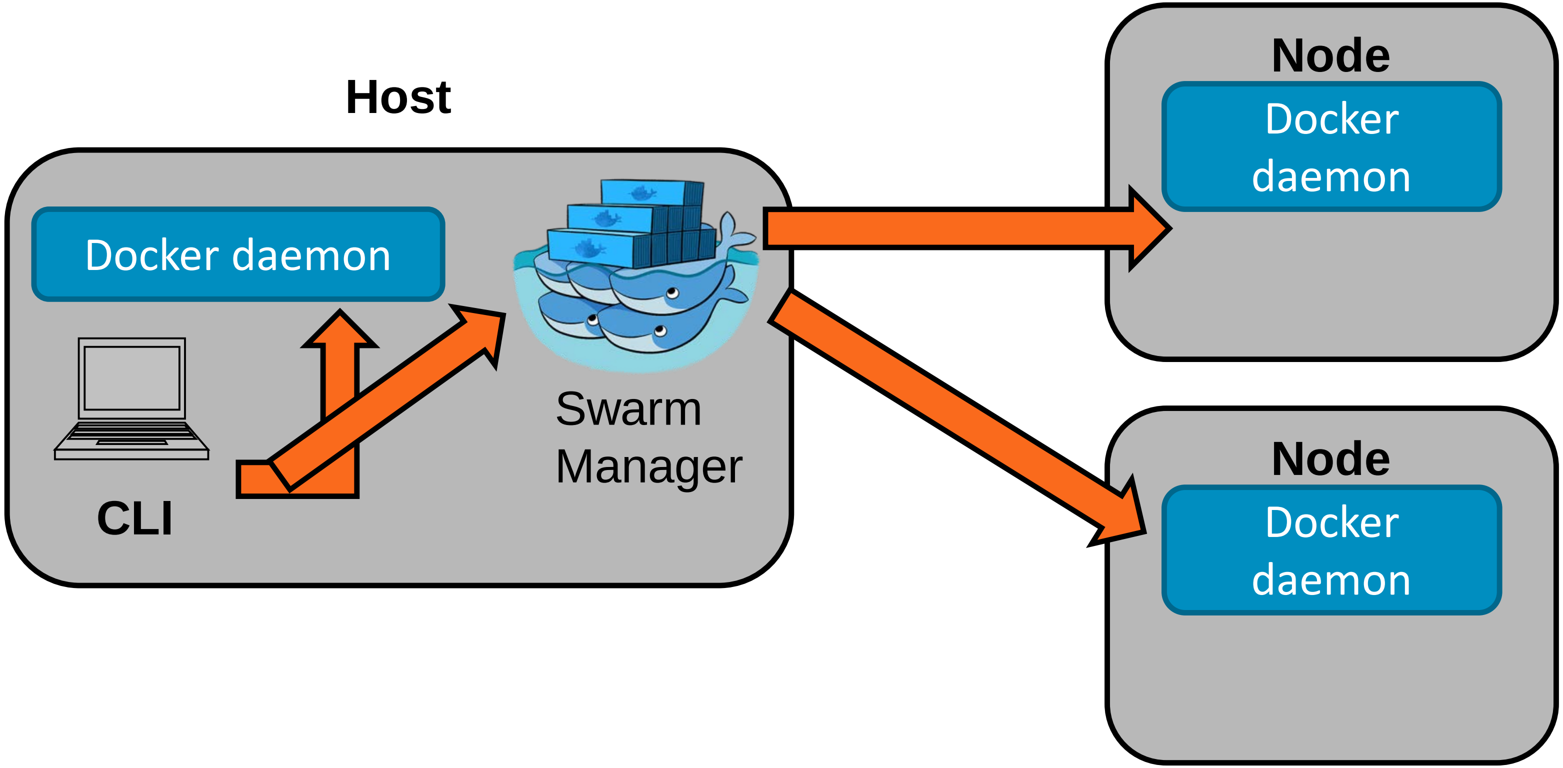
What is Docker Swarm?

***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

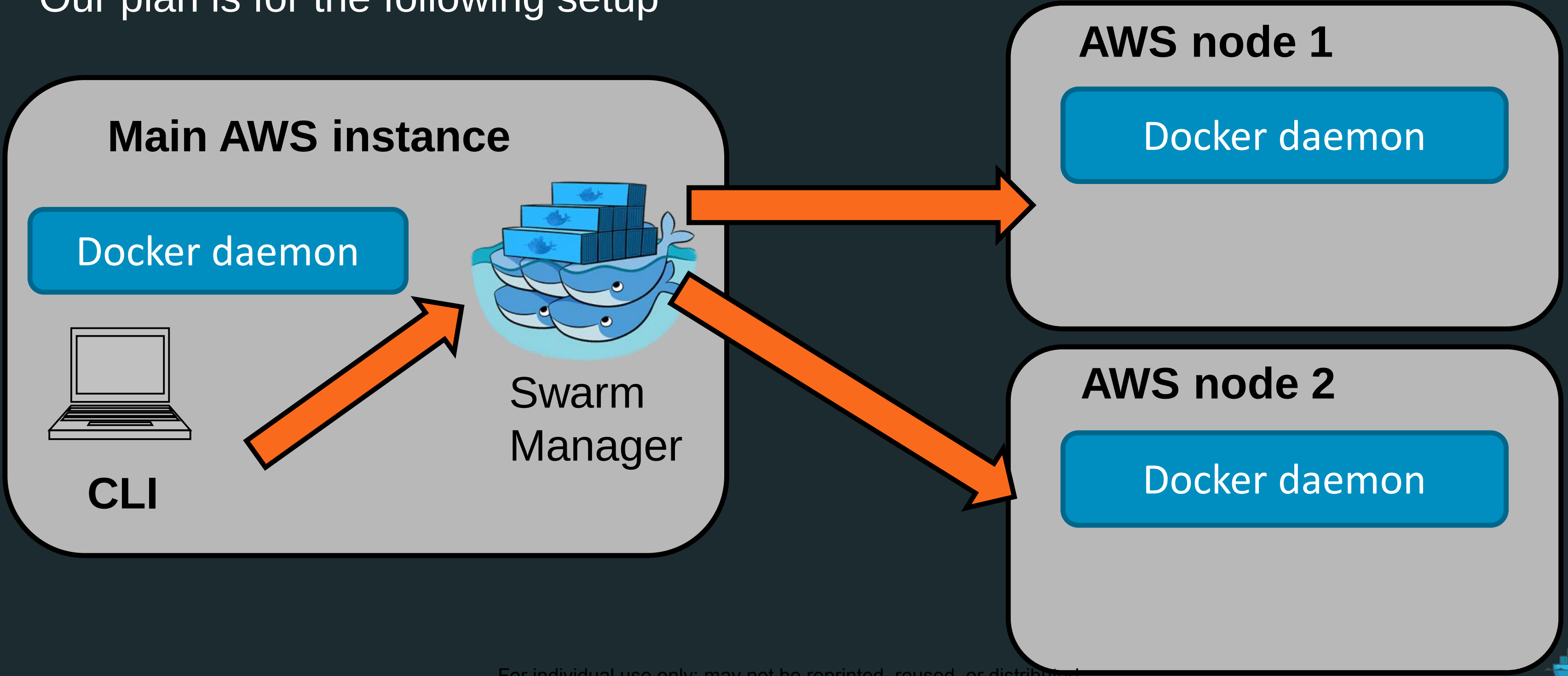
- There are two ways to install Swarm
  - Install the Swarm binary
  - Run Swarm in a container using the Swarm image
- The Swarm binary needs to be installed on every machine that is going to be part of the cluster.
- For instructions on installing the binary see https://github.com/docker/swarm
- Using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

EX8.1 – Start swarm manager

EXERCISE

Using main AWS instance

1. First, create the token that we will use to identify our cluster
   `docker run --rm swarm create`
2. Copy the output from the command in 1) and paste into a file called `swarmtoken`. Do not lose this token you will need it for the next few exercises. Put the `swarmtoken` file on your home directory
3. Start the swarm manager using automatic port mapping
   `docker run -d -P swarm manage token://$(cat swarmtoken)`

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>
```

EX8.2 – Join nodes to cluster

EXERCISE

1. **Switch to your second AWS instance. This instance will become node 1 in the cluster**
2. Configure the `/etc/default/docker` file to have the daemon listening on tcp://0.0.0.0:2375
   ```
   DOCKER_OPTS="-H tcp://0.0.0.0:2375"
   ```
3. Restart the Docker service
   ```
   sudo service docker restart
   ```
4. Run `ifconfig` and note down the IP address of the machine
5. Join the node to the cluster using the cluster token from your main AWS instance. Make sure you specify the IP address of this host
   ```
   docker run –d swarm join
            --addr=<node ip>:2375 \
            token://<cluster token>
   ```

EX8.2 (cont’d)

EXERCISE

6. **Switch back to your first AWS instance and list the nodes in the cluster**
   `docker run --rm swarm list token://<cluster_token>`
7. Verify you can see the IP address of your 2nd AWS instance on the list
8. **Now repeat the same steps on your third AWS instance. This instance will become node 2 in the cluster**
9. By the end of this exercise you should have your Swarm master managing two nodes

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
- Inspect at the container port mapping to find the Swarm host IP and port
  - In our example, the Swarm is at IP 172.17.0.2

Configure the DOCKER_HOST variable

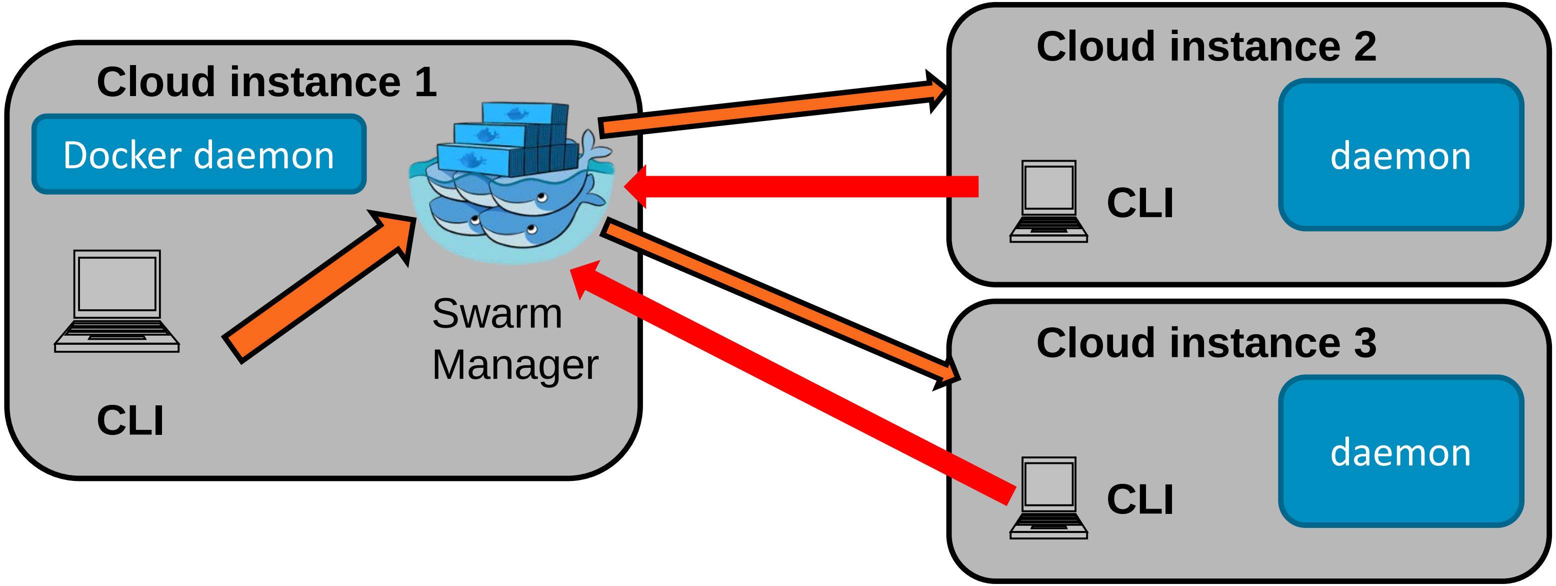
```
export DOCKER_HOST=172.17.0.2:<swarm port>
```

Run docker client and specify the daemon to connect to

```
docker –H tcp://172.17.0.2:<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

```
Client:
 Version:      1.10.2
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   c3959b1
 Built:        Mon Feb 22 21:37:01 2016
 OS/Arch:      linux/amd64

Server:
 Version:      swarm/1.1.3
 API version:  1.22
 Go version:   go1.5.3
 Git commit:   7e9c6bd
 Built:        Wed Mar  2 00:15:12 UTC 2016
 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
```

EX8.3 – Connect client

EXERCISE

1. Make sure you are using your main AWS instance
2. Run `docker ps` to find your Swarm container name and note down the port mapping
3. Set the `DOCKER_HOST` variable to the IP of 127.0.01 and specify the port from question 2).
   `export DOCKER_HOST=<ip>:<port>`
4. Run `docker version` and verify that Swarm is listed as the server version
5. Run `docker info` to check the status of your connected nodes

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 of    ...   80/tcp, 443/tcp                    node1/goofy_morse
6387c6790ce8    nginx:latest    "nginx -g 'daemon of    ...   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 is was previously running on

```
CONTAINER ID    IMAGE           COMMAND                  ...   PORTS                                NAMES
a46d77a60121    nginx:latest    "nginx -g 'daemon of     ...   80/tcp, 443/tcp                      node1/goofy_morse
6387c6790ce8    nginx:latest    "nginx -g 'daemon of     ...   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

EX8.4 – Run containers in the cluster

EXERCISE

Using your master AWS instance

1. Run an NGINX container in detached mode
   `docker run -d nginx`
2. What node did Swarm run the container on? (Use `docker ps` to find out)
3. Run a Tomcat container in detached mode
   `docker run -d tomcat`
4. What node is the container running in?

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

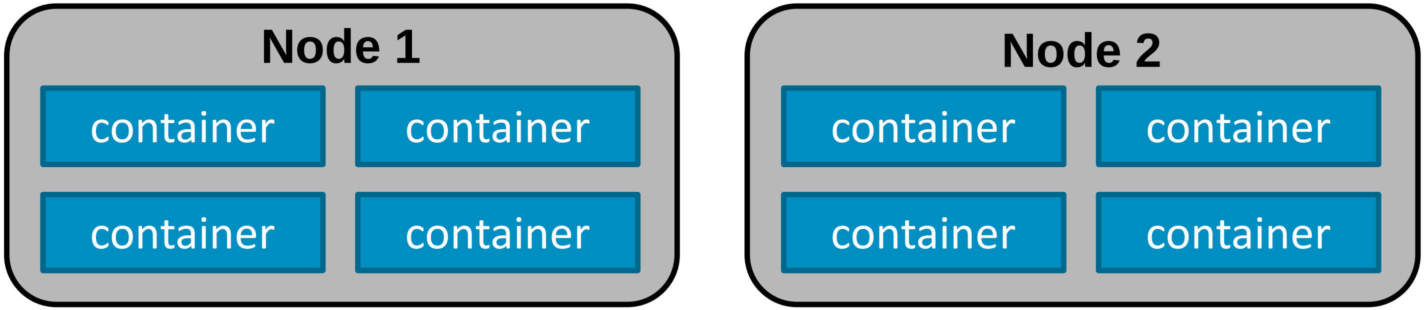
- In your previous exercise, you would have noticed that the two containers you ran, were spread across both nodes
- In this section, we will learn how the Swarm manager decides which node your container runs on

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



EX8.5 – Spread strategy

EXERCISE

Using your master AWS instance

1. Run `docker ps` to check the number of containers you have and what node they are running on. At this stage there should be 1 container on each node
2. Run two more NGINX containers in detached mode and observe which node they are scheduled on
3. Switch over to the terminal on node 1

Using node 1

4. Connect the Docker client to the daemon via TCP
   `export DOCKER_HOST=“tcp://localhost:2375”`
5. Run two more NGINX containers in detached mode. There should now be 4 containers on the host machine
6. **Switch back to the master AWS instance**

EX8.5 – Spread Strategy

EXERCISE

Using your master AWS instance

7. Run `docker info` and to check the number of containers on each node. You should observe 4 containers on one of your nodes and two on the other
8. Run two more NGINX containers in detached mode. What node were they scheduled on?
9. **Quick clean up:** Stop all containers and then delete them

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

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

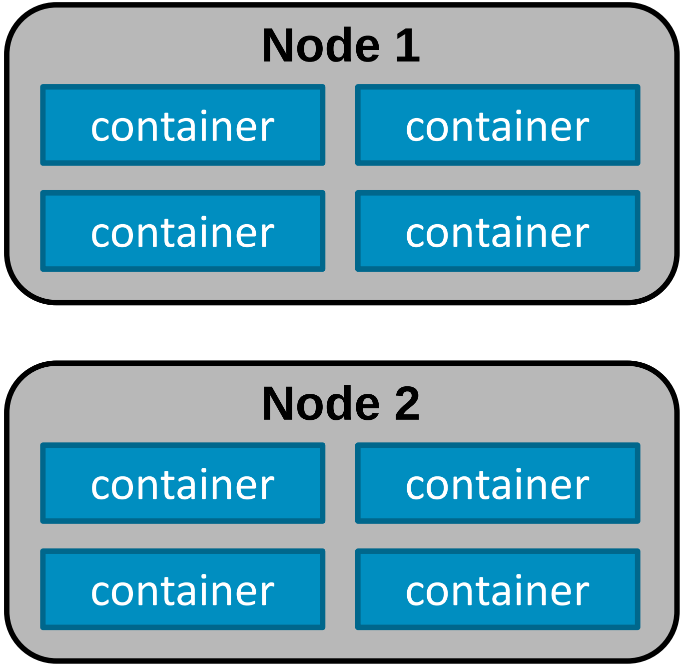
```
docker run -d -m 512MB nginx
```

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
```

EX8.6 – Switch to binpack strategy

EXERCISE

Using master AWS instance

1. Disconnect your Docker client from the Swarm manager
   `unset DOCKER_HOST`
2. Stop the container which is running Swarm
3. Start another container running the `swarm manage` process but specify the `binpack` strategy. Remember to use the same cluster token you created earlier and to specify the port mapping

```
docker run -d -P swarm manage \
         token://$(cat swarmtoken) \
         --strategy binpack
```

4. Run `docker ps` to find out the container name, and which host port has been mapped to the TCP port 2375 on the container
5. Set the `DOCKER_HOST` variable to connect the Docker client to Swarm
   `export DOCKER_HOST=127.0.0.1:<port>`

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
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 514.5 MiB
 node2: 104.236.163.107:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 514.5 MiB
 node3: 104.131.142.17:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ 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
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 200 MiB / 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
```

EX8.7 – Use binpack strategy

EXERCISE

Using master AWS instance

1. Run `docker info` to check the number of containers on each node. There should zero containers at the moment
2. Run two NGINX containers in detached mode
3. Use `docker info` again to check where the containers have been scheduled. Notice they are on the same node
4. Run three more NGINX containers and check which node they are scheduled on. You should observe that all your containers are on the same node
5. Stop all containers and then delete them
```
docker stop $(docker ps -q)
docker rm $(docker ps -aq)
```

EX8.7 – (cont’d)

EXERCISE

6. Now run an NGINX container but specify a memory limit of 200mb
   `docker run -d -m 200mb nginx`
7. Check which node the container has been scheduled on
8. Repeat 6) and check that the container has been scheduled on the same node as 7)
9. Repeat 6) again and verify that this new container is running on your other node.
10. **Clean up:** Stop and delete all containers from the cluster

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 loose 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

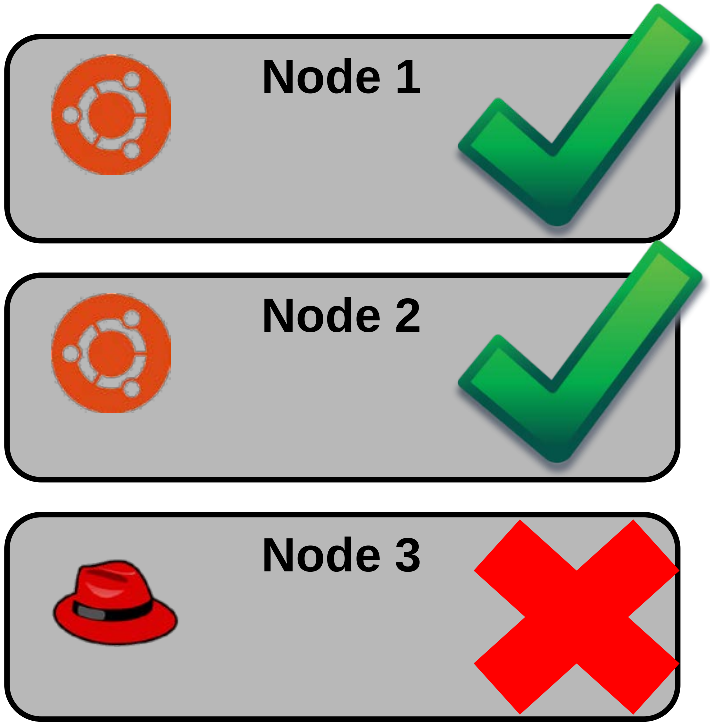
- In the next section we will look at using filters to control how Swarm schedules the containers we run
- We will cover three types of filters
  - Constraint
  - Affinity
  - Port

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
```

EX8.8 – Label the Docker daemon

EXERCISE

Using node 1

1. Open the `/etc/default/docker` file and add the `--label` flag to DOCKER_OPTS. Specify a key value of region=us
   ```
   DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 -H tcp://0.0.0.0:2375 --label region=us”
   ```
2. Restart the Docker service
3. Run `docker info` to confirm the label is present

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
```

EX8.9 – Specify constraint filter

EXERCISE

Using master AWS instance

1. Disconnect the Docker client from Swarm
   `unset DOCKER_HOST`
2. Stop the container which is running the Swarm manage process
3. Start a new container to run the Swarm manage process and use the `spread` scheduling strategy.
   `docker run -d -P swarm manage token://$(cat swarmtoken)`
4. Run an NGINX container and specify the constraint filter of region=us
   `docker run -d -e constraint:region==us nginx`
5. Verify that the container runs on node 1.

EX8.9 – (cont'd)

EXERCISE

6. Repeat step 4 and once again confirm that the container runs on node 1
7. **Clean up:** Stop and delete all containers

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
```

EX8.10 – Container affinity

EXERCISE

Using main AWS instance

1. Run a Tomcat container and give it the name of “appserver”
   `docker run -d --name appserver tomcat`
2. Check which node the container is scheduled on
3. Now run an NGINX container and specify a container affinity so that it is scheduled on the same node as the “appserver” container
   `docker run -d -e affinity:container==appserver nginx`
4. Verify that the NGINX container is running on the same node as the Tomcat container
5. **Clean up:** stop and delete all containers

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
```

EX8.11 – Port filtering

EXERCISE

1. Run an NGINX container and map the container port 80 to port 80 on the host
   `docker run -d -p 80:80 nginx`
2. Check which node the container is scheduled on
3. Repeat 1). This time you should see that the container is running on the other node, because port 80 on the first node has been taken.
4. Repeat 1) again. This time you should notice that Swarm refuses to schedule the container. Why is that so?
5. Run another NGINX container but this time, map the container port 80 to port 8080 on the host. Swarm will schedule this because we have not used up port 8080 on either of our nodes
   `docker run -d -p 8080:80 nginx`
6. **Clean up:** Stop and delete all containers

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

- 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 3: Multi-host 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
2a0797f00bacb6308f733f04cb85983e91d4ba48e09760fbc08abe445856e6ef
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"                                     node2/client
536f7a4708ea        nginx               "nginx -g 'daemon off"     80/tcp, 443/tcp      node1/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 you 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

Rescheduling containers

- When a node fails, Swarm can reschedule the containers on that node to another node
- Containers must be run with the reschedule policy enabled
- Container reschedules are recorded in the Swarm manager log

Run an nginx container with a rescheduling policy in the event of node failure

```
docker run -d --env="reschedule:on-node-failure" nginx
```

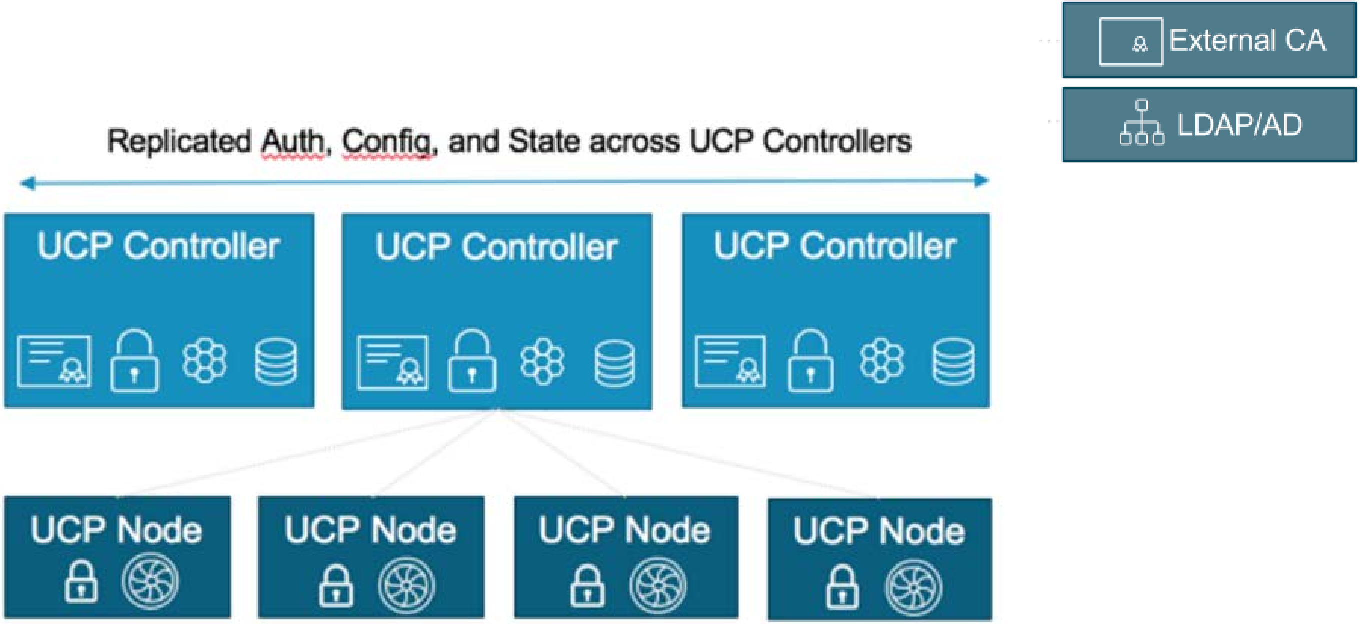
Universal Control Plane

- In this section
  - Introduction to the UCP
  - UCP Architecture
  - Deploying a container with UCP

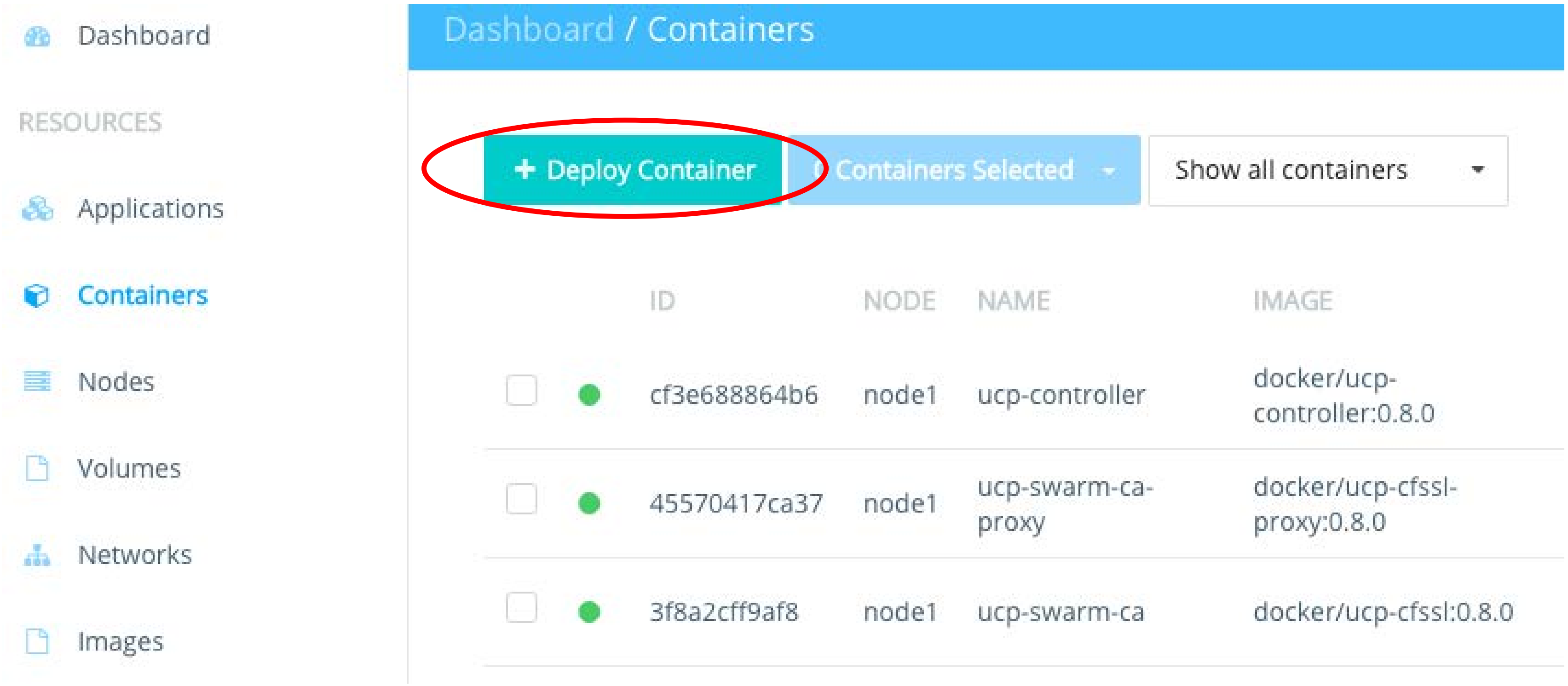
Introduction to the UCP

- Universal Control Plane is a Docker native solution designed to provision and cluster Docker hosts and their resources.
- Distributed via a bootstrapped container-based installer available from Docker Hub, similar to Swarm
- UCP simplifies the container and application deployment process using Swarm, making it more accessible to users not as intimately familiar with Docker
- Targeted at large enterprise data centre environments

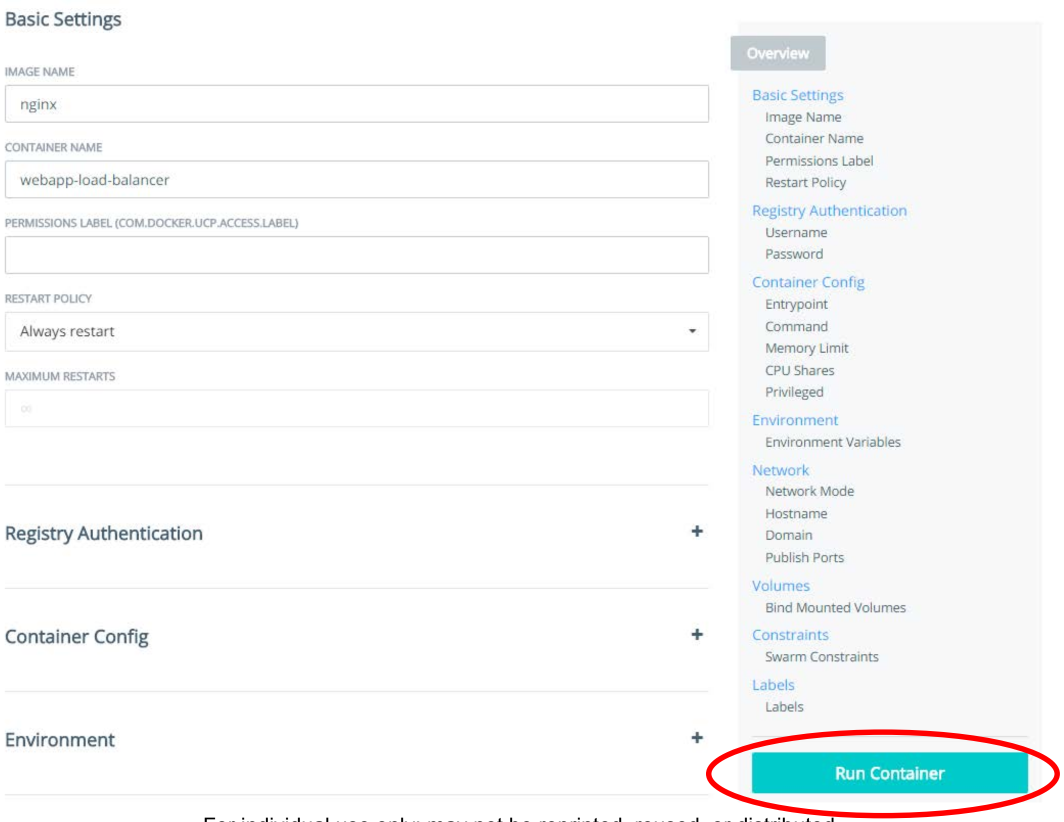
UCP Architecture



Deploying a container using the UCP



Deploying a container using the UCP



UCP + Swarm

- UCP ships with extra features on top of Swarm targeted at an enterprise data-centre environment
- Role-based permissions management system on top of the local Swarm Manager, which can help manage your containers
- Native support for Content Trust image signing
- TLS enabled by default for UCP webapp and Swarm manage

Module summary

- The way to set up a Swarm cluster differs depending on your chosen discovery backend
- The Swarm scheduler can be configured either the spread or binpack strategy
- The spread strategy distributes containers evenly in the cluster
- The binpack strategy schedules as many containers as possible on the same host
- Filtering is applied before the scheduling of containers in order to eliminate unsuitable nodes
- Constraint filters are based on the labels applied to your Docker daemon and / or certain fields from the `docker info` command

Module Summary (cont'd)

- Affinity filters allow us Swarm to schedule a container on the same node as another container or a node with a specific image available
- Port filtering allows Swarm to only consider nodes where the specified port is available
- Docker UCP can be used to simplify the setup and maintenance of a Swarm cluser

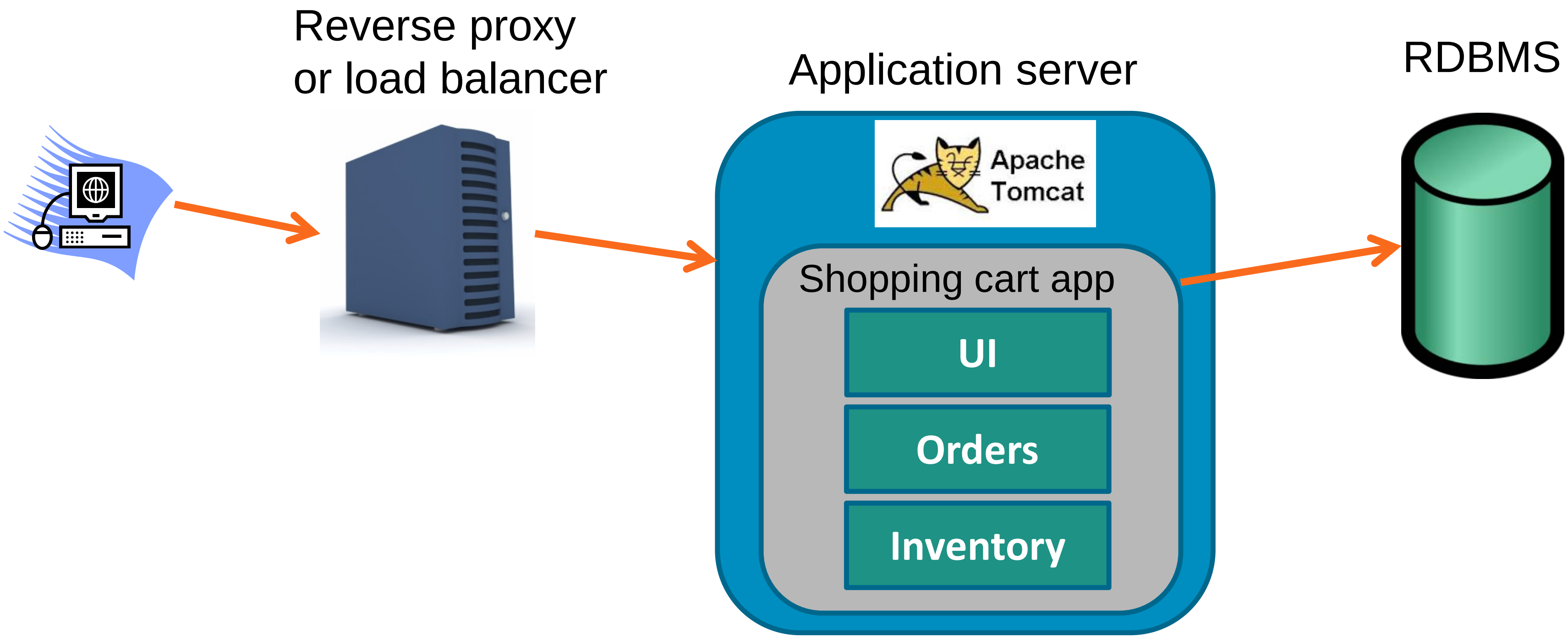
Module 9:
Building Micro Service Applications

Module objectives

In this module we will:

- Understand how Docker containers can help in a micro service application architecture
- Build an example of linking containers and see how it can potentially be used in a micro service application architecture

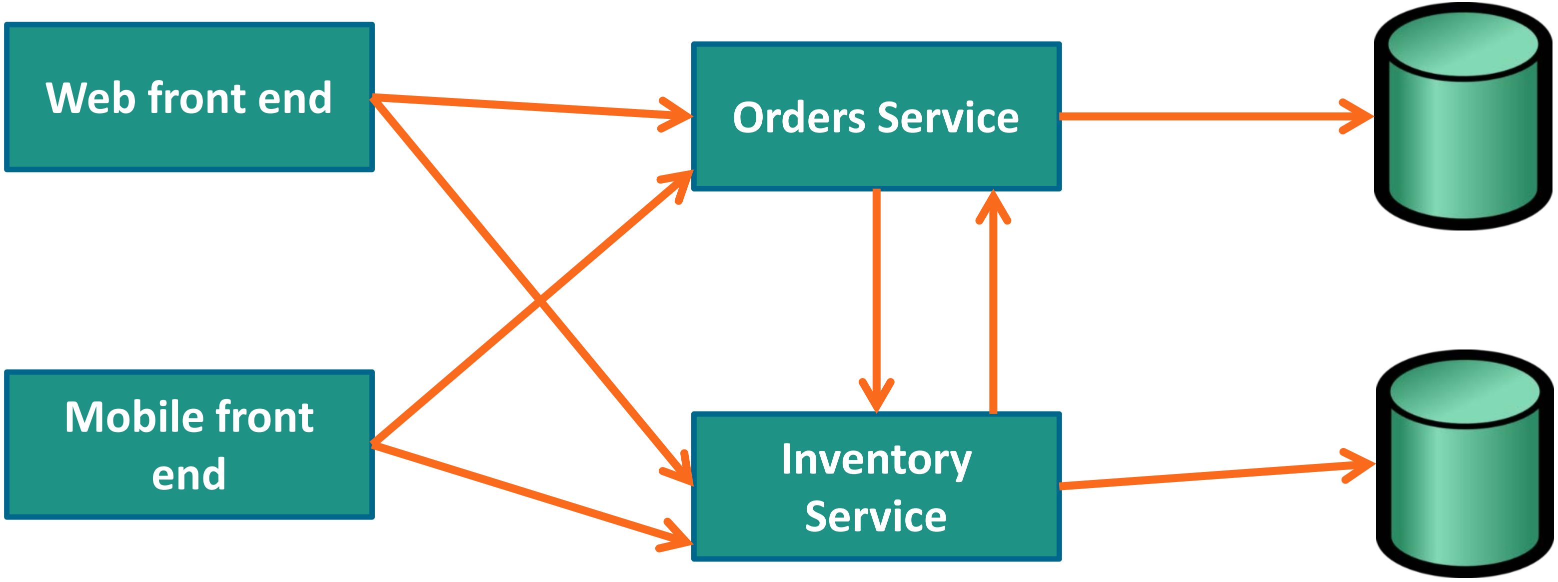
Traditional style monolithic architecture



Weaknesses with monolithic structure

- Hard to upgrade each component individually. Have to upgrade the entire stack.
- Tied to a particular technology stack.
  - E.g. If you start with Java and Spring MVC framework you are stuck with it.
  - Difficult to migrate to newer technology.
- If one component goes down, the entire application can go down. No one can visit your website
- Overloaded IDE
- Not ideal for scaling because you have to scale the entire stack

Micro service architecture



Main principles

- Each service should have one concern and only focus on that concern
- Services should be able to run independently of each other
- Each service component should be independently upgradable and replaceable
- Each service should be lightweight and quick to develop
- Services interact with each other by exposing an API

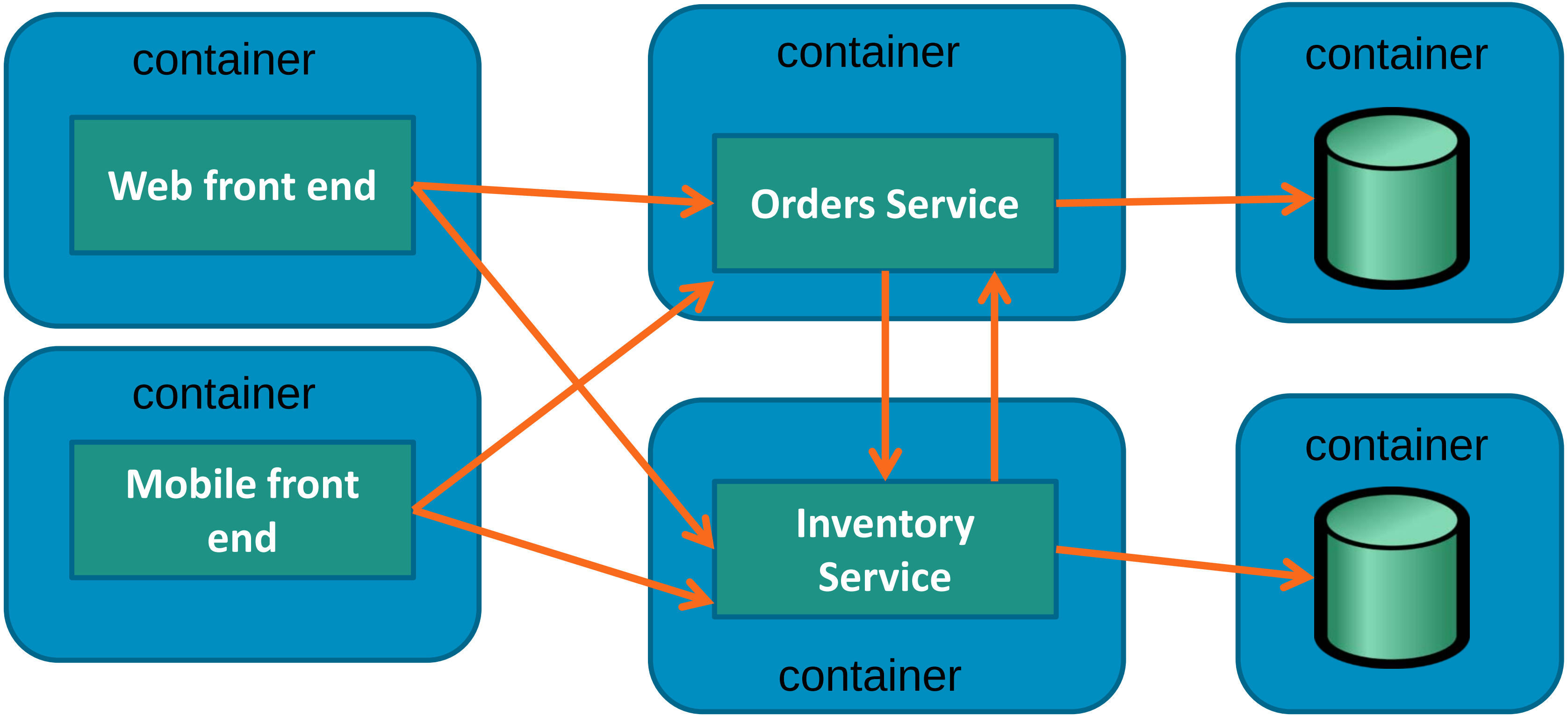
Advantages

- Each service can be developed and upgraded independently
- Easier for developers to understand
  - Only have to focus on their service
- If one service goes down, the application should still run, albeit with reduced functions
- Application is easier to troubleshoot
- The whole application does not have to be committed to one technology stack

How Docker fits in

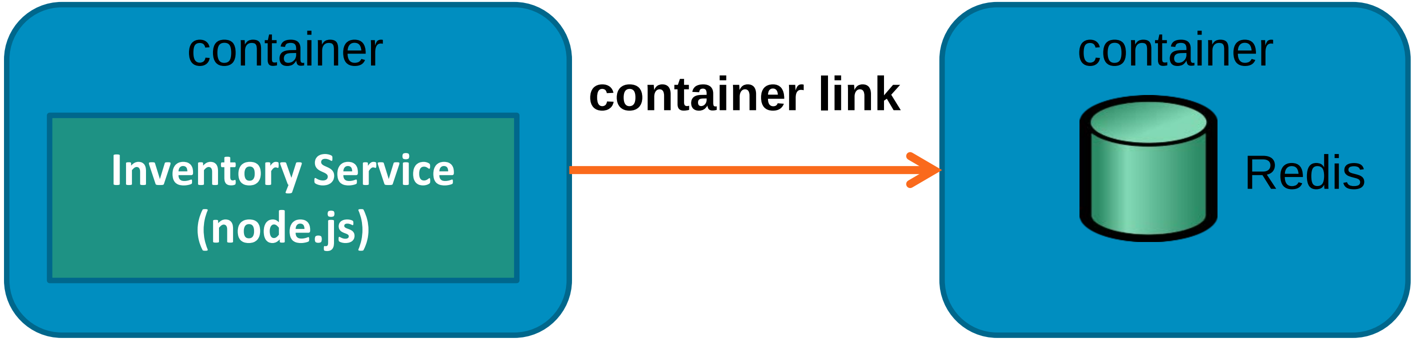
- Each service runs on its own container
- Container links can be used to connect services together or connect a service with it's database
  - For example: the inventory service connects to another container running the database, as opposed to having the database in the same container
- Docker Compose can be used to strap the containers together to ensure they are linked and launch all the containers to start the entire application

Micro service architecture with containers



Example to build

- We will build a simple example of a node.js application linked to a Redis database
- Let’s pretend our node.js application is the inventory service



Process

- We will start with a standalone Node.js application and connect it to Redis, which will be running in a container
- Then we will dockerize the app and run it in a container
- We will link the container to our Redis container and the application should be able to establish it's connection to the Redis server

EX9.1 - Install Node.js and sample code

EXERCISE

1. Install Node.js on your AWS instance
   `sudo apt-get install nodejs`
2. Install the Node.js package manager
   `sudo apt-get install npm`
3. In your home folder, check out the source code of our application
   `git clone https://github.com/johnny-tu/inventory-service.git`
4. Change directory into the inventory-service folder
5. Install the node_redis package using the Node package manager
   `npm install redis`

Our sample code

inventoryService.js

```
var redis = require('redis');
var client = redis.createClient();

client.on('connect', function() {
    console.log('connected');
});

client.set("books_count", "123", redis.print);
client.get("books_count", function (err, reply) {
    console.log('books_count = ' + reply.toString());
});
```

EXERCISE

EX9.2 - Run the code

1. Run a Redis server inside a container. Use the `redis` image and map the container port 6379 to host port 6379. Remember to run in detached mode. Give the container the name “`redisdb`”
   `docker run -d -p 6379:6379 --name redisdb redis`
2. Run the Node.js application
   `nodejs inventoryService.js`
3. Verify that you see the following output

```
johnnytu@docker-ubuntu:~/nodeexample$ nodejs inventoryService.js
connected
Reply: OK
books_count = 123
```

Dockerize our Node.js application

- The Dockerfile has been included for your convenience

```
FROM node:0.12.4

COPY inventoryService.js /src/
WORKDIR /src

RUN npm install redis
CMD ["node", "inventoryService.js"]
```

EX9.3 – Dockerize the application

EXERCISE

1. Build an image from the given Dockerfile. Name your image repository inventory-service
   `docker build -t inventory-service .`
2. Create a new bridge network called nodeapp
   `$ docker network create nodeapp`
3. Stop and remove the existing `redis` container and start a new one on the nodeapp network
4. Run a container from your built image and link it to the Redis container
   `docker run --net nodeapp inventory-service`
5. Notice the error message saying that it can’t connect to 127.0.0.1:6379. This is because our code doesn’t take into account the link we established.

EX9.3 – (cont'd)

EXERCISE

6. Open the inventoryService.js file and change the line
   `var client = redis.createClient();` into
   `var client = redis.createClient(6379, "redisdb");`
7. Build the image again
   `docker build -t inventory-service .`
8. Repeat step 2. Run a container from your built image and link it to the Redis container.
   `docker run --net nodeapp inventory-service`
9. This time you should see the output of the application

Module summary

- A micro service application architecture has many advantages over a traditional monolithic architecture
- Docker containers make it easier to deploy a micro service application

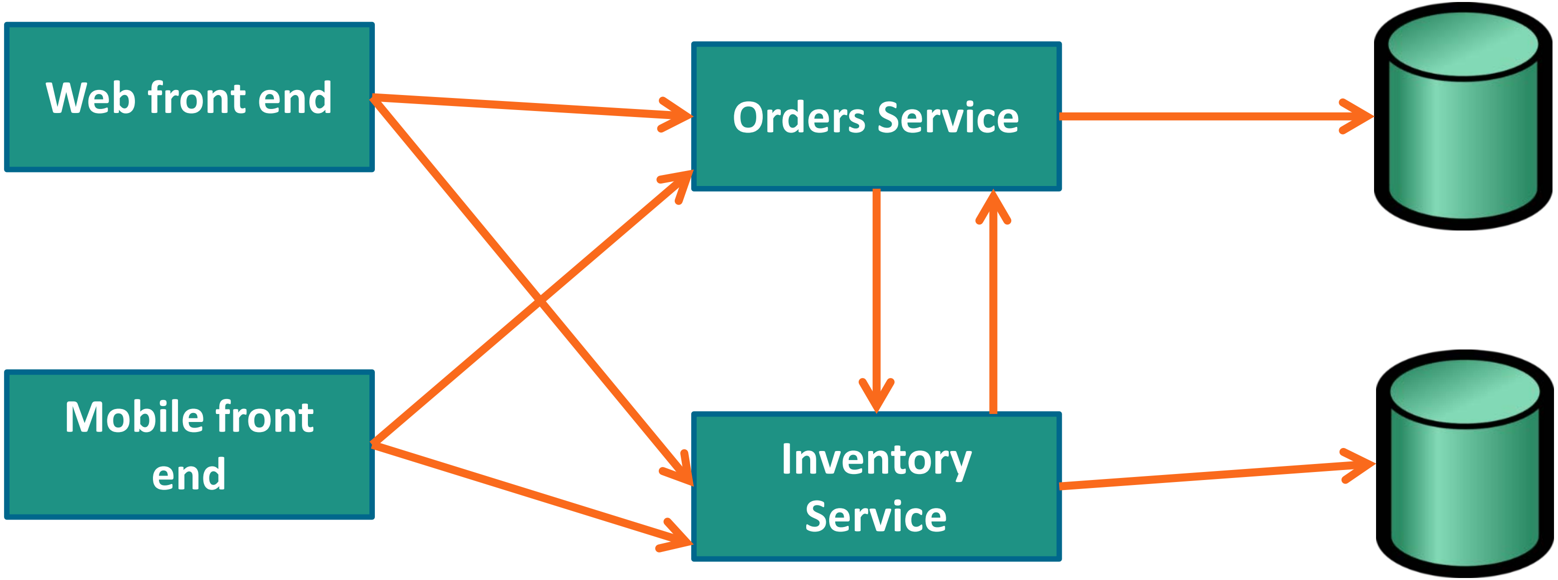
Module 10: Docker Compose

Module objectives

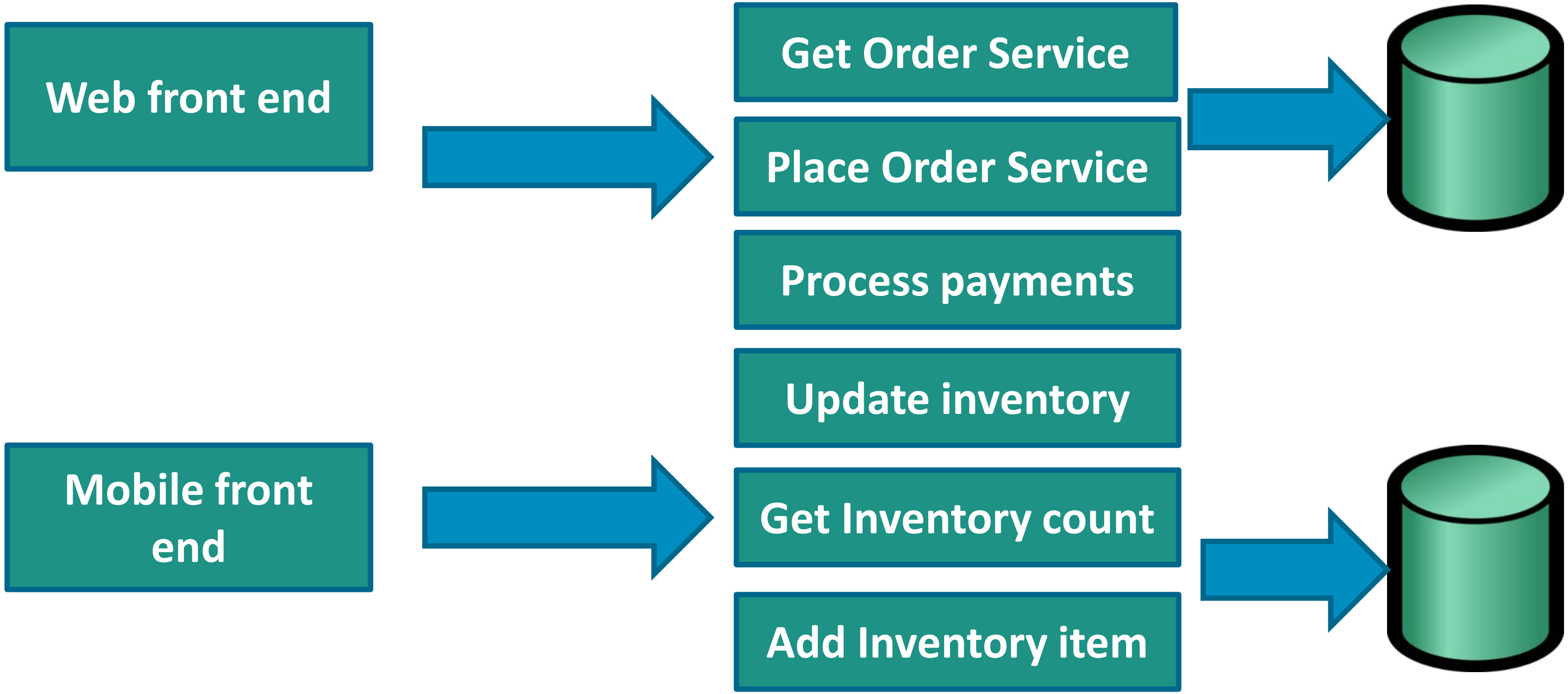
In this module we will:

- 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



Web front end
Mobile front end
Get Order Service
Place Order Service
Process payments
Update inventory
Get Inventory count
Add Inventory item

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

EX10.1 – Install Docker Compose

EXERCISE

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

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

Use the latest redis Image from Docker Hub

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
```

EX10.2 – Try the example manually

EXERCISE

1. In your home directory, download the sample code
   ```
   git clone \
   https://github.com/johnny-tu/HelloRedis.git
   ```
2. Build the image using the Dockerfile. Name your image "`helloredis`"
   ```
   docker build -t helloredis .
   ```
3. Create a new bridge network called `java_app`
   ```
   docker network create –d bridge java_app
   ```
4. Run a Redis container called `redisdb` on your `java_app` network
   ```
   docker run -d --name redisdb --net java_app redis
   ```
5. Run your `helloredis` image on the `java_app` network container. Verify that the command line output tells you are connected to Redis. Remember to specify a name for the container
   ```
   docker run ––name client ––net java_app helloredis
   ```

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

```
johnnytu@docker-ubuntu:~/HelloWorld$ docker-compose up
Recreating helloworld_redis_1...
Recreating helloworld_javaclient_1...
Attaching to helloworld_redis_1, helloworld_javaclient_1
redis_1       | 1:C 30 May 13:25:50.147 # Warning: no config file specified, using the default config. In order to specify a config
file use redis-server /path/to/redis.conf
redis_1       |                 _._
redis_1       |            _.-``__ ''-._
redis_1       |       _.-``    `.  `_.  ''-._           Redis 3.0.1 (00000000/0) 64 bit
redis_1       |   .-`` .-```.  ```\/    _.,_ ''-._
redis_1       |  (    '      ,       .-`  | `,    )     Running in standalone mode
redis_1       |  |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
redis_1       |  |    `-._   `._    /     _.-'    |     PID: 1
redis_1       |   `-._    `-._  `-./  _.-'    _.-'
redis_1       |  |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1       |  |    `-._`-._        _.-'_.-'    |           http://redis.io
redis_1       |   `-._    `-._`-.__.-'_.-'    _.-'
redis_1       |  |`-._`-._    `-.__.-'    _.-'_.-'|
redis_1       |  |    `-._`-._        _.-'_.-'    |
redis_1       |   `-._    `-._`-.__.-'_.-'    _.-'
redis_1       |       `-._    `-.__.-'    _.-'
redis_1       |           `-._        _.-'
redis_1       |               `-.__.-'
redis_1       |
redis_1       | 1:M 30 May 13:25:50.152 # Server started, Redis version 3.0.1
redis_1       | 1:M 30 May 13:25:50.152 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition
. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_mem
ory=1' for this to take effect.
```

EX10.3 – Compose our application

EXERCISE

1. Shut down your existing redis container
2. Run the example application
   `docker-compose up`
3. Check the output on your terminal. You should see the output of the Redis server starting up, followed by the output of our Java client
4. Hit `CTRL + C` to terminate the services

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

```
johnnytu@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                Up
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

EX10.4 – Running in detached mode

EXERCISE

1. Remove the two services from the previous exercise
   `docker-compose rm`
2. Start the example application again but this time, run in detached mode
   `docker-compose up -d`
3. Run `docker-compose ps` and check that both the `javaclient` and `redis` service are running
4. Check the log for the `javaclient`
   `docker-compose logs javaclient`
5. Stop the `javaclient` service
   `docker-compose stop javaclient`
6. Start the `javaclient` service again
   `docker-compose start`

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
```

EX10.5 – Scaling services

EXERCISE

1. Scale the `javaclient` service to run 5 containers
   `docker-compose scale javaclient=5`
2. Check your containers with `docker-compose ps`
3. View the aggregated log of your `javaclient` service
   `docker-compose logs javaclient`
4. Pick any one of the `javaclient` containers and display the log output for just that container
   `docker logs <container name>`

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 an 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. It's 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

- 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 based on the project name
- In order to handle the the revised Docker networking system, a new Compose yml format was defined, named Version 2

Compose yml Version 2

- Compose V1 files continue to utilise the older container-linking system and do not support networking.
- We have been using the Compose V1 format as it is still in common use on many existing projects at this point
- For any new projects, it is recommended to use Version 2 only
- Differences are relatively minor in many cases
- You can easily identify a v2 file – it will have a `version: '2‘` entry at the top of the file

Using the bridge network with Compose

- Since all containers in a bridge network can lookup other containers using their name, there is no need to define links in your configuration file
- Let's consider the following configuration example
  - Rename the old docker-compose.yml to docker-compose.v1.yml, and replace it with this version

docker-compose.yml

```
version: '2‘
services:
  javaclient:
    build: .
  redis:
    image: redis
```

Using the bridge network with Compose

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

```
student@masterhost:~/HelloRedis$ docker-compose 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_default": {
                "IPAMConfig": null,
                    "Links": null,
                    "Aliases": [
                        "javaclient",
                        "8aa13e1785"
                    ],
                "NetworkID": "2e1084dede3625b49e0469f6bcf71d928d9ebf4bc587343d36f8b8c4bb199084",
                "EndpointID": "045fbabe6fd7ee7f63522de9b995dbf73d86210abdfe8dbce2fa58f88153ab4f",
                "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 validate that we can ping our first `redis` container from `helloredis_javaclient_1`

```
student@masterhost:~/HelloRedis$ docker exec -it
helloredis_javaclient_1 ping helloredis_redis_1

Pinging helloredis_redis_1 [172.19.0.2] with 32 bytes of
data:
Reply from 172.19.0.2: bytes=32 time=8ms TTL=56
Reply from 172.19.0.2: bytes=32 time=8ms TTL=56
…
```

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

```
version: '2'
services:
  javaclient:
    build: .
  redis:
    image: redis
    container_name: redisdb
```

EX10.6 – Compose and Networking

EXERCISE

1. In the HelloRedis example, rename docker-compose.yml file to docker-compose.v1.yml. Remake docker-compose.yml.
2. In the `redisdb` service, add a `container_name` instruction and name the container `redisdb`
   `container_name: redisdb`
3. Your docker-compose.yml file should look like this:

```
version: '2'
 services:
   javaclient:
     build: .
   redis:
     image: redis
     container_name: redisdb
```

EX10.6 – (cont'd)

EXERCISE

4. Run `docker-compose up -d`
5. Run `docker network ls` and verify that you can see a bridge network named `helloredis_default`
6. Inspect the newly created containers and verify that they are on the `helloredis_default` network
   ```
   docker inspect helloredis_javaclient_1
   docker inspect redisdb
   ```
7. Check the javaclient log and make sure it is able to ping the redisdb container. The output should say:
   ```
   Server is running: PONG
   books_count = null
   ```

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
- As we have learned in multi-host networking, containers running on different nodes cannot communicate with each other unless the nodes are part of an overlay network

Compose and overlay networks

- Compose normally 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
- When you run Compose apps on a Swarm Manager, it will automatically try and create 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
- As long as this is set up, you can just run your app normally

Example

```
docker-compose 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
```

EX10.7 – Compose on Swarm

EXERCISE

1. Build the image defined in the Dockerfile of the HelloRedis application folder. Tag the image with your Docker Hub account name
   `docker build -t <docker hub login>/hello-redis:1.0 .`
2. Push the image to Docker Hub
   `docker push <docker hub login>/hello-redis:1.0`
3. Open the `docker-compose.yml` file.
4. Modify the `javaclient` service by replacing the `build` parameter with an image parameter and specify the image you've just built and pushed
   `image: <docker hub login>/hello-redis:1.0`

EX10.7 – (cont'd)

EXERCISE

5. Connect the Docker client to Swarm
6. Now run `docker-compose up -d`.
7. Run `docker ps` and check that your two containers are on different nodes
8. Check the logs to your `helloredis_javaclient_1` container. You will notice that it fails to connect to `redisdb` because the redis container is on a different node
   ```
   docker logs helloredis_javaclient_1
   ```
9. Stop and remove the services
   ```
   docker-compose stop
   docker-compose rm -f
   ```

EX10.7 – (cont'd)

EXERCISE

10. Run the application again but this time, use the compose networking option. It will default to the overlay network driver if possible.
    `docker-compose up -d`
11. Run `docker network ls` and check that you have an overlay network named `helloredis_default`
12. Inspect the javaclient container and verify that it is running on the `helloredis_default overlay network`
    `docker inspect helloredis_javaclient_1`
13. Check the `javaclient` container log and verify that it can reach the `redisdb` container.
    `docker logs helloredis_javaclient_1`
14. The logs output should be the following two lines repeated constantly
    ```
    Server is running: PONG
    books_count = null
    ```

EX10.7 – (cont'd)

15. Now let's scale our `javaclient` service to run 5 containers
    `docker-compose scale javaclient=5`
16. Run `docker ps` to check the containers and verify that the `javaclient` containers are spread across both Swarm nodes

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

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