Module 6:
Docker Machine

docker

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

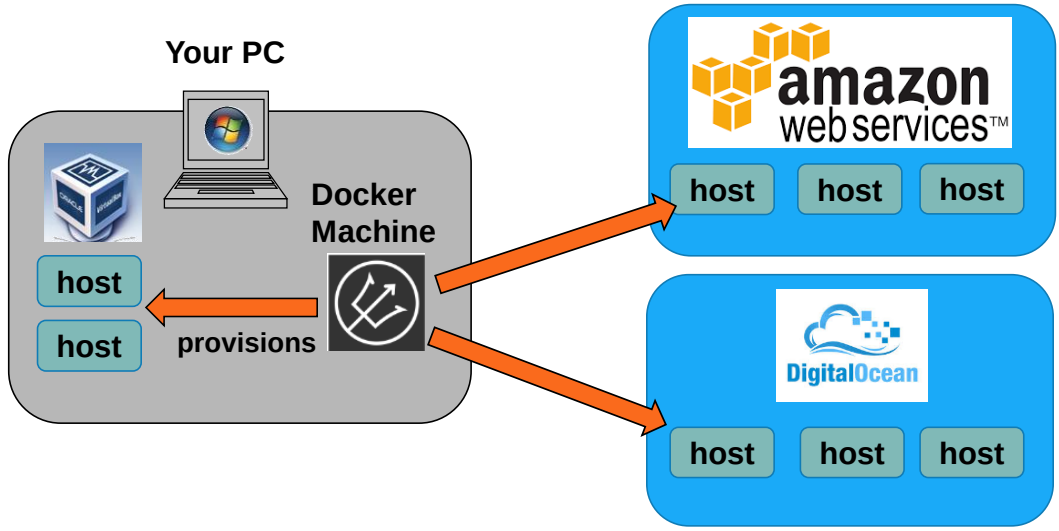
Docker Machine overview

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

Docker Machine overview



docker

Installing Machine

- Go to https://github.com/docker/machine/releases and follow the command line instructions for your operating system
- Alternatively
  - Download the binary release for your operating system at https://github.com/docker/machine/releases
  - Rename binary to `docker-machine`
  - Place binary into a folder of your choice (recommended folder is /usr/local/bin for Linux)
  - Add the folder to your system environment PATH
  - Set appropriate permissions

docker

Installing machine on Linux

- Download the latest Linux binary from https://github.com/docker/machine/releases
- Rename binary to `docker-machine`
- Place binary into a folder of your choice (recommended folder is /usr/local/bin for Linux)
- Set appropriate permissions

```
$ sudo chmod 755 /usr/local/bin/docker-machine
```


docker

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

docker

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>
```


docker

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

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

docker

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

```
ubuntu@node-0:~$ docker-machine ls
NAME        ACTIVE   DRIVER         STATE     URL                         SWARM   DOCKER    ERRORS
testhose2   -        digitalocean   Running   tcp://104.236.43.223:2376           v1.12.1
testhost1   -        digitalocean   Running   tcp://104.131.82.247:2376           v1.12.1
```


docker

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

docker

Setting the active host

```
ubuntu@node-0:~$ docker-machine ls
NAME           ACTIVE   DRIVER         STATE     URL                         SWARM   DOCKER    ERRORS
jtucloudhost1  -        digitalocean   Running   tcp://104.131.98.26:2376            v1.12.1
jtucloudhost2  -        digitalocean   Running   tcp://104.131.79.248:2376           v1.12.1
ubuntu@node-0:~$
ubuntu@node-0:~$ eval $(docker-machine env jtucloudhost1)
ubuntu@node-0:~$ docker-machine ls
NAME           ACTIVE   DRIVER         STATE     URL                         SWARM   DOCKER    ERRORS
jtucloudhost1  *        digitalocean   Running   tcp://104.131.98.26:2376            v1.12.1
jtucloudhost2  -        digitalocean   Running   tcp://104.131.79.248:2376           v1.12.1
```


docker

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 /
```

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
```


docker

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
```


docker

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…

```
ubuntu@node-0:~$ docker-machine inspect jtucloudhost1
{
    "ConfigVersion": 3,
    "Driver": {
        "IPAddress": "104.131.98.26",
        "MachineName": "jtucloudhost1",
        "SSHUser": "root",
        "SSHPort": 22,
        "SSHKeyPath":
"/home/ubuntu/.docker/machine/machines/jtucloudhost1/id_rsa",
        "StorePath": "/home/ubuntu/.docker/machine",
        "SwarmMaster": false,
        "SwarmHost": "tcp://0.0.0.0:3376",
        "SwarmDiscovery": "",
```


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

```
ubuntu@node-0:~$ docker-machine ip jtucloudhost1
104.131.98.26
ubuntu@node-0:~$ ping $(docker-machine ip jtucloudhost1)
PING 104.131.98.26 (104.131.98.26) 56(84) bytes of data.
64 bytes from 104.131.98.26: icmp_seq=1 ttl=47 time=93.4 ms
64 bytes from 104.131.98.26: icmp_seq=2 ttl=46 time=93.1 ms
64 bytes from 104.131.98.26: icmp_seq=3 ttl=46 time=93.0 ms
64 bytes from 104.131.98.26: icmp_seq=4 ttl=47 time=93.1 ms
```

Module 7:
Building Micro Service Applications

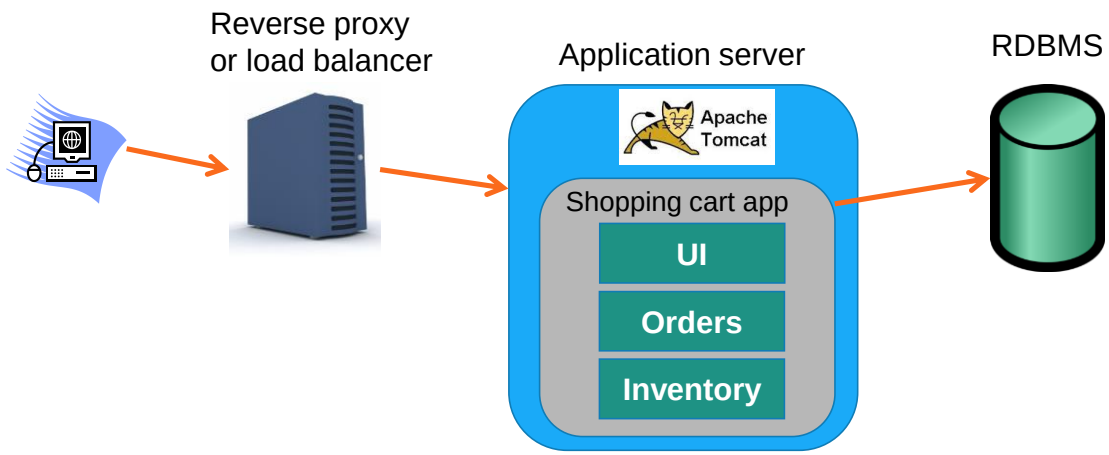
Module objectives

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

docker

Traditional style monolithic architecture

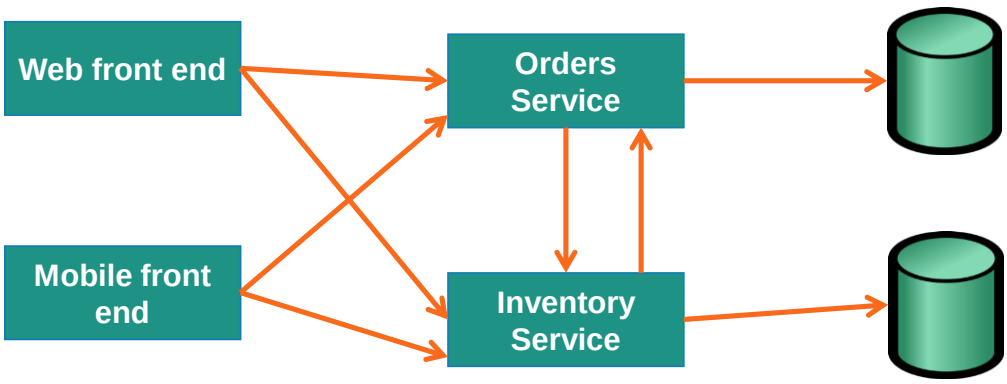


docker

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

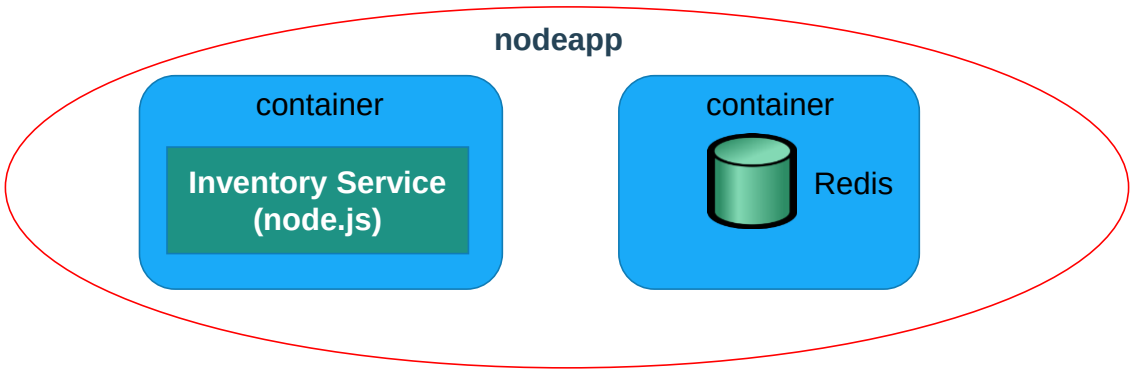
- Each service should have one concern and only focus on that concern
- Services should be able to run independently of each other
- Each service component should be independently upgradable and replaceable
- Each service should be lightweight and quick to develop
- Services interact with each other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

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
- Both containers will run on a custom bridge network called **nodeapp**



docker

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

docker

EX7.3 – Dockerize the application

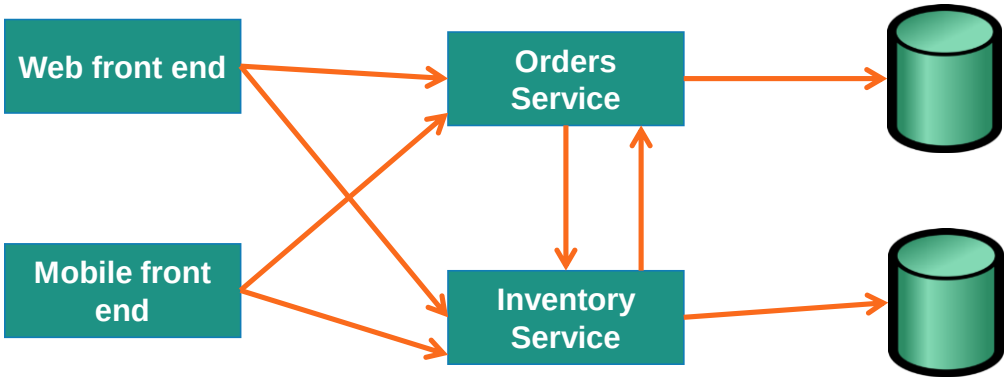
EXERCISE

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
   ```
   $ docker network create mynetwork1
   ```
3. Run a Redis container
   ```
   docker run --net mynetwork1 --name myredis1 redis
   ```
4. Run a container from your built image and link it to the Redis container
   ```
   docker run --net mynetwork1 --name myapp inventory-service
   ```

docker

Module 8:
Docker Compose

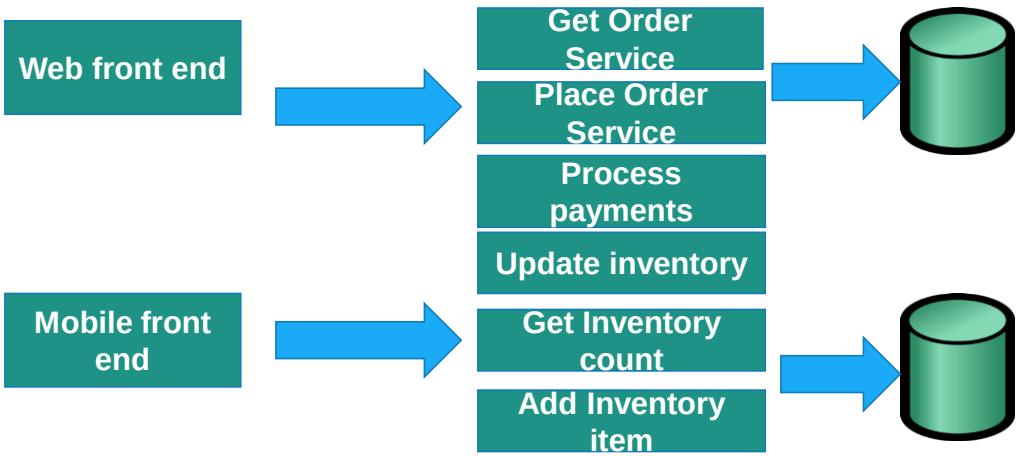
Recall simple micro service example



353

docker

The reality



354

docker

Compose file format version

- *Particularly relevant if you have used Compose before...*
- Compose 1.6 introduced support for a new Compose file format (aka "v2")
- Services are no longer at the top level, but under a services section
- There has to be a version key at the top level, with value "2" (as a string, not an integer)
- Containers are placed on a dedicated network, making links unnecessary
- There are other minor differences, but upgrade is easy and straightforward

v1 vs v2 compose file

Version 1

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

Version 2

```
version: '2'

services:
  javaclient:
    build: .
    command: java HelloWorld
  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
```


docker

Image instruction

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

Use the latest redis Image from Docker Hub

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


docker

Links, naming and service discovery

- Containers can have network aliases (resolvable through DNS)
- Compose file version 2 makes each container reachable through its service name
- Compose file version 1 requires "links" sections
- Our code can connect to services using their short name
- (instead of e.g. IP address or FQDN)

Example code

```
16
17  redis = Redis("redis")
18
19
20  def get_random_bytes():
21      r = requests.get("http://rng/32")
22      return r.content
23
24
25  def hash_bytes(data):
26      r = requests.post("http://hasher/",
27                        data=data,
28                        headers={"Content-Type":
```

Running your application

- Use **`docker-compose up`**
- Up command will
  - Build the image for each service
  - Create and start the containers
- Displays aggregated logs of all the containers (when running in foreground mode)
- Containers can all run in the foreground or in detached mode

Running your application

```
ubuntu@node-0:~/orchestration-workshop/dockercoins$ docker-compose up
Starting dockercoins_webui_1
Starting dockercoins_redis_1
Starting dockercoins_rng_1
Starting dockercoins_worker_1
Starting dockercoins_hasher_1
Attaching to dockercoins_hasher_1, dockercoins_rng_1, dockercoins_redis_1, dockercoins_worker_1, dockercoins_webui_1
redis_1    |
redis_1    |
redis_1    |                                      Redis 3.2.3 (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 08 Aug 11:32:40.053 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/co
ower value of 128.
redis_1    | 1:M 08 Aug 11:32:40.054 # Server started, Redis version 3.2.3
redis_1    | 1:M 08 Aug 11:32:40.054 # WARNING overcommit_memory is set to 0! Background save may fail under low memory co
dd 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for
redis_1    | 1:M 08 Aug 11:32:40.055 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This
ry usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled'
/etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
redis_1    | 1:M 08 Aug 11:32:40.055 * The server is now ready to accept connections on port 6379
worker_1   | INFO:__main__:0 units of work done, updating hash counter
hasher_1   | == Sinatra (v1.4.7) has taken the stage on 80 for development with backup from Thin
worker_1   | ERROR:__main__:In work loop:
worker_1   | Traceback (most recent call last):
```

EX8.3 – Compose our application

EXERCISE

1. Run the dockercoins application
   `docker-compose up`
2. Check the output on your terminal. You should see the output of the Redis server starting up, followed by the output of the other containers
3. Hit `CTRL + C` to terminate the services

docker

Our application log output

- The application continuously generates logs
- We can see the worker service making requests to rng and hasher
- CTRL + C stops all containers by sending them the TERM signal
- Some containers exit immediately, others take longer (because they don't handle SIGTERM and end up being killed after a 10s timeout)

docker

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`

docker

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
```


docker

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**

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`

docker

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>`

docker

View service container logs

- Use `docker-compose logs` command
- If a service is not specified, the aggregated log of all containers will be displayed
- Use `--follow` option to follow the log output
  - `CTRL + S` to pause following
  - `CTRL + Q` to resume following
- Use `--tail` option to start following logs from last "x" number of lines
- Example (follow logs from last 10 lines)

```
docker-compose logs --tail 10 --follow
```


docker

EX8.4 – Running in detached mode

EXERCISE

1. Start the **dockercoins** application again but this time, run in detached mode

```
$ docker-compose up -d
```

2. Run `docker-compose ps` and check that all services are running
3. View all logs since container creation and exit when done

```
$ docker-compose logs
```

4. View all logs for the `redis` service

```
$ docker-compose logs redis
```

5. Follow the logs from the last 20 lines for the `rng` service

```
$ docker-compose logs --follow --tail 20 rng
```


docker

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
```


docker

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 worker`
  Will display the aggregated log output of all `worker` containers
- To run a command against just one individual container, use standard Docker commands and specify the container name
  `docker logs dockercoins_worker_2`

```
ubuntu@node-0:~/orchestration-workshop/dockercoins$ docker-compose ps
        Name                      Command               State          Ports
-------------------------------------------------------------------------------------
dockercoins_hasher_1   ruby hasher.rb                   Up      0.0.0.0:8002->80/tcp
dockercoins_redis_1    docker-entrypoint.sh redis ...   Up      6379/tcp
dockercoins_rng_1      python rng.py                    Up      0.0.0.0:8001->80/tcp
dockercoins_webui_1    node webui.js                    Up      0.0.0.0:8080->80/tcp
dockercoins_worker_1   python worker.py                 Up
dockercoins_worker_2   python worker.py                 Up
```


docker

EX8.7 – Scaling services

EXERCISE

1. Scale the `worker` service of the dockercoins application to run 2 containers
   `docker-compose scale myredis=3`
2. Check your containers with `docker-compose ps`
3. View the aggregated log of your `redis` service
   `docker-compose logs myredis`

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`
  - `ports`
- Some other parameters include
  - `volumes` – for defining volumes
  - `command` – specifying which command to execute
- Full reference list at https://docs.docker.com/compose/yml/

docker

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

docker

Yml parameters and Dockerfile instructions

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

Volume handling

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

Docker Compose build

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

docker

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}
  …
  …
```


docker

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

docker

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

docker

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

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
```

Ex 8.9 - Cleanup

EXERCISE

1. Tell Docker Compose to remove everything
   ```
   docker-compose down
   ```

Module 9:
Swarm Mode

docker

Module objectives

In this module we will:

- Explain the concepts of Swarm mode in Engine 1.12
- Setup and manage a Swarm cluster
- Deploy, manage and scale applications in our Swarm cluster
- Outline the multi host networking functions in the Swarm cluster

docker

Swarm Mode – Part 1: Overview and concepts

docker

Introduction to SwarmKit

- SwarmKit is an open source toolkit to build multi-node systems
- It is a reusable library, like libcontainer, libnetwork, vpnkit ...
- It is a plumbing part of the Docker ecosystem
- SwarmKit comes with two examples:
  - swarmctl (a CLI tool to "speak" the SwarmKit API)
  - swarmd (an agent that can federate existing Docker Engines into a Swarm)
- See the project repository at https://github.com/docker/swarmkit

docker

SwarmKit key features

- Cluster management using the Docker Engine.
  - No need for additional orchestration software.
- Highly-available, distributed store based on Raft Consensus Algorithm
- *Services* managed with a *declarative API* (implementing *desired state* and *reconciliation loop*)
- Automatic TLS keying and signing
- Dynamic promotion/demotion of nodes, allowing to change how many nodes are actively part of the Raft consensus
- Service discovery, integration with overlay networks and load balancing
- Full features list at https://docs.docker.com/engine/swarm/

Orchestration with SwarmKit

- Desired state reconciliation
  - Example: On node failure, tasks are rescheduled onto a different node
- Replicated services vs Global services
- Configurable updates
- Restart policies
- Scaling

Docker Engine Swarm Mode

- Docker Engine 1.12 features SwarmKit integration
- The Docker CLI features three new commands:
  - `docker swarm` (enable Swarm mode; join a Swarm; adjust cluster parameters)
  - `docker node` (view nodes; promote/demote managers; manage nmodes)
  - `docker service` (create and manage services)
- The Docker API exposes the same concepts
- The SwarmKit API is also exposed (on a separate socket)

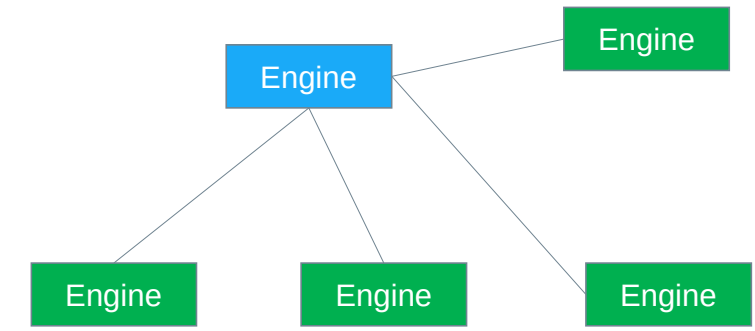
docker

Key concepts – Swarm and Node

- Swarm
  - A cluster of Docker Engines where services are deployed
  - Engines participating in a cluster are running in **swarm mode**
  - Docker Engine CLI includes commands for swarm management, service deployment and orchestration
- Node
  - An instance of a Docker engine participating in a Swarm
  - Can be a **manager** node or **worker** node
  - Managers are also workers

docker

Illustration of concepts

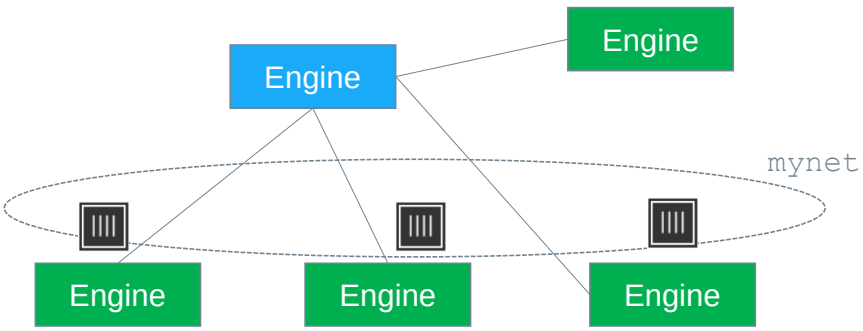



```
$ docker swarm init
```

```
$ docker swarm join <IP of manager>:2377
```


docker

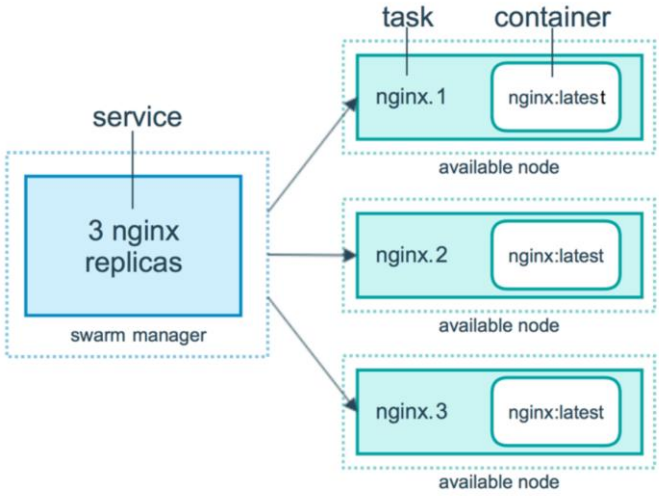
Creating services




```
$ docker service create --replicas 3 --name myapp
--network mynet --publish 80:80 myapp:1.0
```


docker

Services, tasks and containers



435

docker

Swarm mode – Part 2: Creating our swarm

docker

Creating our first Swarm

- By default, everything runs as usual
- Swarm Mode can be enabled, "unlocking" SwarmKit functions (services, out-of-the-box overlay networks, etc.)
- The cluster is initialized with `docker swarm init`
- This should be executed on a first, seed node
- **DO NOT** execute `docker swarm init` on multiple nodes!
  - You would have multiple disjointed clusters.

Under the hood

- When we run `docker swarm init`, the Engine performs the following
  - switches the current node into swarm mode.
  - creates a swarm named `default`.
  - designates the current node as a leader manager node for the swarm.
  - names the node with the machine hostname.
  - configures the manager to listen on an active network interface on port 2377.
  - sets the current node to `Active` availability, meaning it can receive tasks from the scheduler.
  - starts an internal distributed data store for Engines participating in the swarm to maintain a consistent view of the swarm and all services running on it.
  - by default, generates a self-signed root CA for the swarm.
  - by default, generates tokens for worker and manager nodes to join the swarm.
  - creates an overlay network named `ingress` for publishing service ports external to the swarm.

Token generation

- In the output of `docker swarm init`, we have a message confirming that our node is now the (single) manager:
- Docker also generated two security tokens (like passphrases or passwords) for our cluster, and shows us the commands to use on other nodes to add them to the cluster using those security tokens:

```
ubuntu@node-0:~$ docker swarm init
Swarm initialized: current node (9dlewwkhaqbonjucmg4rxj9pr) is now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-5bpm5ddm5at6abswm7s4fa9zk41x31lb3kj7aaqrhv6j7a4ahb-atkxvn26ru2i2g7yn2f69pidj \
    10.0.9.219:2377

To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-5bpm5ddm5at6abswm7s4fa9zk41x31lb3kj7aaqrhv6j7a4ahb-6lojw9j9srjmvhbgztuz7k8d0 \
    10.0.9.219:2377
```


docker

Checking that Swarm mode is enabled

- Run `docker info` and look for the following:

```
Swarm: active
 NodeID: 9dlewwkhaqbonjucmg4rxj9pr
 Is Manager: true
 ClusterID: 0t1tez69maryzfpzoz6zwnedo
 Managers: 1
 Nodes: 1
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot interval: 10000
  Heartbeat tick: 1
  Election tick: 3
 Dispatcher:
  Heartbeat period: 5 seconds
 CA configuration:
  Expiry duration: 3 months
```


docker

EX 9.1 – Create our swarm

EXERCISE

Using your master node

1. Try the following Swarm-specific command and notice how we get an error:
   `$ docker node ls`
2. Create a swarm cluster
   `$ docker swarm init`
3. Run `docker info` to check that the swarm has been enabled
4. Now run `docker node ls` again and check that the output shows one node in the cluster

Adding nodes to the Swarm

- We need the token that was shown earlier
- You wrote it down, right?
- Don't panic, we can easily see it again 😉
- Use the `docker swarm join-token worker` command

```
ubuntu@node-0:~$ docker swarm join-token worker
To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-
5bpm5ddm5at6abswm7s4fa9zk41x31lb3kj7aaqrhv6j7a4ahb-
atkxvn26ru2i2g7yn2f69pidj \
    10.0.9.219:2377
```

Check our swarm nodes

- Run the `docker node ls` command
- Will display all nodes that are part of the cluster
- Must run command on a manager node
- Only manager nodes can accept swarm specific commands

```
ubuntu@node-0:~$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER
STATUS
1fcmolxnoeomgj9j1vzk1hxak    node-1    Ready   Active
9dlewwkhaqbonjucmg4rxj9pr *  node-0    Ready   Active        Leader
```


docker

EX 9.2 – Add nodes

EXERCISE

Using your master node

1. Show the Swarm join command
   `$ docker swarm join-token worker`
2. Open a new terminal and SSH into **node-1**

Using node-1

3. Copy paste the `docker swarm join` ... command
   (that was displayed just before)
4. Try and run `docker node ls`.
   Notice the error message saying that “This node is not a swarm manager”

docker

EX 9.2 – (cont'd)

EXERCISE

Switch back to your master node terminal

5. Run `docker node ls` and verify that you can see **node-1** in the list
6. Now repeat the sames steps but for **node-2** and **node-3**

By the end of this exercise, you should have a 4 node cluster with the master node (or node-0) as Leader

```
ubuntu@node-0:~$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
1fcmolxnoeomgj9j1vzk1hxak    node-1    Ready   Active
44h62hedvf0z86as6hq3q36i8    node-3    Ready   Active
9dlewwkhaqbonjucmg4rxj9pr *  node-0    Ready   Active        Leader
b7s890mhn77dn2h2psdec8z51    node-2    Ready   Active
```


Under the hood - certificates

- When we do `docker swarm init`, a TLS root CA is created. Then a keypair is issued for the first node, and signed by the root CA.
- When further nodes join the Swarm, they are issued their own keypair, signed by the root CA, and they also receive the root CA public key and certificate.
- All communication is encrypted over TLS.
- The node keys and certificates are automatically renewed on regular intervals (by default, 90 days; this is tunable with `docker swarm update`).

Adding node as a manager

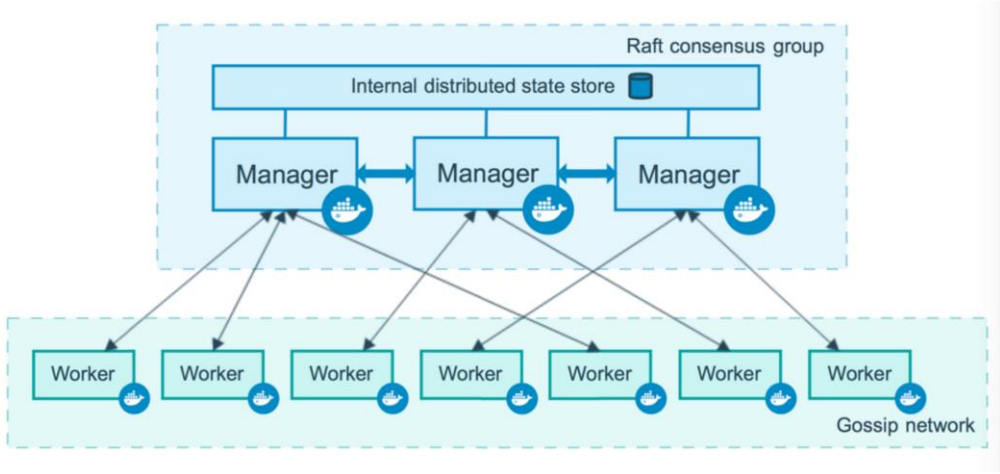
- For a cluster to be highly available, we need more than just one manager node
- Adding a node as manager requires a different token
- Token can be obtained with `docker swarm join-token manager`

```
ubuntu@node-0:~$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-5bpm5ddm5at6abswm7s4fa9zk41x31lb3kj7aaqrhv6j7a4ahb-6lojw9j9srjmvhbgztuz7k8d0 \
    10.0.9.219:2377


ubuntu@node-0:~$ docker swarm join-token worker
To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-5bpm5ddm5at6abswm7s4fa9zk41x31lb3kj7aaqrhv6j7a4ahb-atkxvn26ru2i2g7yn2f69pidj \
    10.0.9.219:2377
```


docker

Nodes overview



docker

Manager nodes

- Maintaining cluster state
- Scheduling services
- Serving swarm mode HTTP API endpoints
- Uses a raft implementation
- How may do I need?
  - A three-manager swarm tolerates a maximum loss of one manager.
  - A five-manager swarm tolerates a maximum simultaneous loss two manager nodes.
  - An N manager cluster will tolerate the loss of at most (N-1)/2 managers.
- Docker recommends a **maximum of** seven manager nodes for a swarm.

Worker nodes

- Purpose is to execute containers
- All managers are also workers by default
  - Can be changed by setting their availability to `Drain`
- Scheduler will not assign tasks to nodes with `Drain` availability

Set the availability of a node-1 to Drain. Existing tasks on the node will gracefully shutdown and be rescheduled on another node

```
docker node update --availability drain node-1
```

Promoting nodes

- Instead of adding a manager node, we can also promote existing workers
- Nodes can be promoted (and demoted) at any time
- Use the `docker node promote` command to promote a worker to manager
- Use `docker node demote` to demote a node from manager to worker

Swarm requirements

- Fixed IP address for manager nodes
  - Workers can use dynamic IP
- Open ports between your hosts
  - TCP port 2377 for cluster management communication
  - TCP and UDP port 7946 for communication among nodes
  - TCP and UDP port 4789 for overlay network traffic

Swarm mode – Part 3:
Creating and managing services

Creating a service

- Use the `docker service create` command
- Similar options to `docker run`
- Specify the image
- Specify various options
  - `--name` for service name
  - `--publish` for port mapping
- `docker service create --help` for full list of options

docker

View all services

- Use the `docker service ls` command to see all services
- `docker service ls -q` to only display the service IDs

```
ubuntu@node-0:~$ docker service create alpine ping 8.8.8.8
4xg147hpqmv0ffoem3sivhgmn

ubuntu@node-0:~$ docker service ls
ID            NAME                    REPLICAS  IMAGE   COMMAND
4xg147hpqmv0  condescending_stallman  1/1       alpine  ping 8.8.8.8
```


docker

Check service tasks

- To view all tasks (and containers) of a service, use the `docker service ps` command
- Must specify the service ID or service name
- Shows which node each task is running on

```
ubuntu@node-0:~$ docker service ps condescending_stallman
ID          NAME                      IMAGE   NODE    DESIRED STATE  CURRENT STATE          ERROR
6bsru6jbs… condescending_stallman.1  alpine  node-0  Running        Running 2 minutes ago
```


docker

Check container logs

- Right now, there is no direct way to check the logs of our container (unless it was scheduled on the current node)
- Look up the NODE on which the container is running (in the output of the `docker service ps` command)
- Then SSH into that NODE
- Use `docker logs` command on the container

docker

Ex 9.4 – Create a simple service

EXERCISE

1. Create a service featuring an Alpine container pinging Google resolvers:
   ```
   $ docker service create alpine ping 8.8.8.8
   ```
2. Check which node the container was created on:
   ```
   $ docker service ps <serviceID>
   ```
3. SSH into that node
4. See that the container is running and check its ID:
   ```
   $ docker ps
   ```
5. View its logs:
   ```
   $ docker logs <container ID>
   ```

Specifying service replicas

- When creating a service, we can specify how many replicas to run
- This creates a desired state where the Swarm will always have "x" amount of containers running for that service
- If a node failure results in one or more containers failing, Swarm will schedule new containers on a different node to ensure that we always have our desired number of replicas
- Use the `--replicas` option when creating a service

Create an nginx service with 3 replicas

```
docker service create --replicas 3 nginx
```


Scaling a service

- Services can be scaled with the `docker service update` command
- What we want is to update the number of replicas
- Syntax:

```
docker service update <service name> \
   --replicas=<number>
```

- The `docker service update` command can also be used to update other settings
- Can also use `docker service scale` command. For example:

```
docker service scale myservice=4
```

Ex 9.5 – Scaling a service

EXERCISE

Using master node (node-0)

1. Scale our Alpine service from Ex 10.4 to ensure we have 2 copies per node: (We should have 4 nodes in our cluster)
   `$ docker service update <service name> --replicas=8`
2. Run `docker service ps <service name>` and check that we have eight tasks in total and that there are 2 on each node

docker

Exposing a service

- Services can be exposed on a host port, with two special properties:
  - the public port is available on *every node of the Swarm*,
  - requests coming on the public port are load balanced across all instances.
- This is achieved with option `-p/--publish`; as an approximation:
- If you indicate a single port number, it will be mapped on a port starting at 30000
  (vs. 32768 for single container mapping)
- You can indicate two port numbers to set the public port number manually

docker

Exposing a service (cont'd)

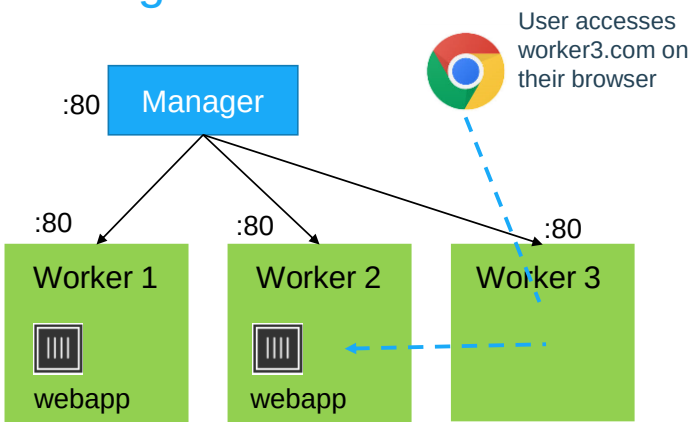
Create an nginx service and expose the container port 80 on host port 8080

```
docker service create --publish 8080:80 nginx
```

Create an nginx service and expose the container port 80 on a host port starting from port 30000

```
docker service create --publish 80 nginx
```

Routing mesh



- Operator reserves a swarm wide ingress port for `webapp`
- Every node listens on port 80
- Container aware routing mesh can transparently route traffic from worker 3 to a node that is running the container
- Built in load balancing
- DNS based service discovery

```
$ docker service create --replicas 2 --publish 80:80
--name webapp webapp:1.0
```


docker

Ex 9.6 – Expose ports

EXERCISE

1. Create an ElasticSearch service called “search”. Specify to run 7 replicas and map the service port 9200 to host 9200
   ```
   $ docker service create --name search \
     --publish 9200:9200 --replicas 7 elasticsearch
   ```
2. Use curl to reach your elastic search service a few times
   ```
   $ curl localhost:9200
   ```

   Each request should be served by a different ElasticSearch instance.

   (You will see each instance advertising a different name.)

docker

Ex 9.6 – (cont'd)

EXERCISE

3. Run an NGINX service and expose the service port 80 on port 8080
   `$ docker service create --name nginx --publish 8080:80 nginx`
4. Check which node your NGINX service task is scheduled on
   `$ docker service ps nginx`
5. Open a web browser and hit IP address of that node with port 8080. You should see the NGINX welcome page
6. Try the same thing with the IP address of any other node in your cluster (using port 8080). You should still be able to see the NGINX welcome page due to the routing mesh

docker

Global services

- A global service is a service where there is one task on every node
- When new nodes are added, the orchestrator will create a new task for the service for that node
- Service is listed as global on the `REPLICAS` column when using `docker service ls`
- Ideal for
  - Monitoring agents
  - Anti virus scanners

Create a global service using the 'myagent' image

```
ubuntu@node-0:~$ docker service create --mode global myagent
14k8go1ici2jymrcjd1ljppf9
ubuntu@node-0:~$ docker service ls
ID            NAME                  REPLICAS  IMAGE          COMMAND
14k8go1ici2j  high_northcutt        global    nginx
```


docker

Deleting services

- Can use `docker service rm` command
  - Specify the service ID or name
  - Accepts multiple services as input
- Shortcut to delete all services
  - `docker service rm $(docker service ls -q)` OR
  - `docker service ls -q | xargs docker service rm`

Ex 9.7 - Cleanup

1. Delete all services so that we can prepare our environment for future exercises

```
$ docker service rm $(docker service ls -q)
```

Swarm mode – Part 4: Deploying applications on Swarm

docker

Shipping images

- When we do `docker-compose up`, images are built for our services
- Those images are present only on the local node
- We need those images to be distributed on the whole Swarm
- The easiest way to achieve that is to use a Docker registry
- Once our images are on a registry, we can reference them when creating our services

docker

Rolling updates

- We want to release a new version of the **`worker`** service
- Process to follow
  - We will edit the code ...
  - build the new image ...
  - push it to the registry ...
  - update our service to use the new image
- Use `docker service update <service> --image <image>` to update to a newer version of the image used for the service

Watching the update deployment process

- Recommended practice to open a new terminal window
- Use `watch` to observe the service status
- If a service has multiple tasks, Docker will update then one at a time
- Can configure parallel updates

Look at our service status

```
watch -n1 "docker service ps worker | grep -v
Shutdown.*Shutdown"
```

Parallel updates

- We can set upgrade parallelism (how many instances to update at the same time)
- And upgrade delay (how long to wait between two batches of instances)
- `--update-parallelism` – how many tasks should be updated at a time
- `--update-delay` – the delay in between update a task
- Both options are used on the `docker service update` command

Change the parallelism to 2 and the delay to 5 seconds:

```
docker service update worker --update-parallelism 2
\
 --update-delay 5s
```

Distributed Application Bundle

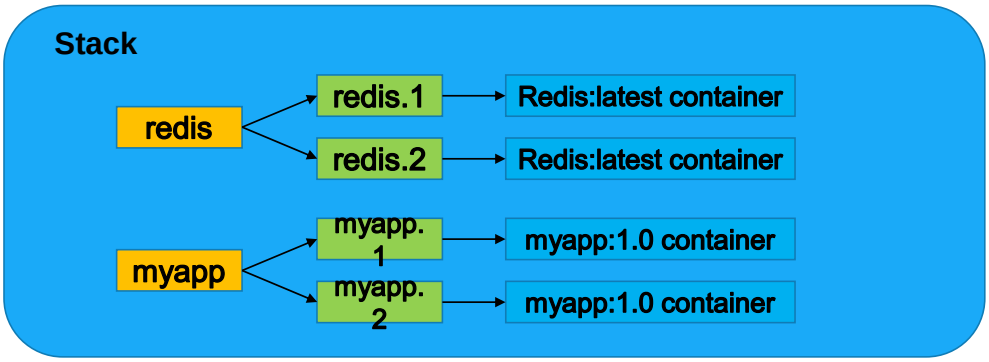
- Real life micro services applications contain hundreds of services
- Manually creating each service to deploy our application isn't ideal
- **Distributed Application Bundle (DAB)**
  - Open file format for building services required to ship and deploy multi container applications
  - Contains a description of all the services and the images required to run them, the ports to expose and the networks used to link the services
  - Defined by a DAB (.dab) file, which can be created from Compose files
- **Note:** feature is currently experimental and not supported !!!

Build and deployment process for DABs

- Developers and CI systems build and test individual service images
- Images are pushed into a registry
- Services are then wrapped into a `.dab` file
- Operations team takes the `.dab` file and deploys the DAB as a **stack**

DABs and Stacks

- A **Stack** is a grouping of services that make up an application
  - Defined by the DAB
  - A Stack is created as a result of deploying the DAB file



docker

DAB requirements

- DABs are still experimental as of Docker Engine 1.12
- To create a DAB file from a compose file, you will require Docker Compose 1.8
- To deploy a DAB, you will need an experimental build of Engine 1.12
- More on experimental builds at https://github.com/docker/docker/tree/master/experimental

To get the latest experimental build of Docker Engine, we run

```
curl -sSL https://experimental.docker.com/ | sh
```


docker

Ex 9.16 – Get experimental engine

EXERCISE

1. Run `curl -sSL https://experimental.docker.com/ | sh` on all your nodes to install the experimental Docker Engine
2. To verify that you have the experimental engine, run `docker --version`. You should see Experimental being indicated
3. Also, run `docker --help`. You should see a new **`deploy`** command on the list of command options

docker

Create a DAB file

- Use the `docker-compose bundle` command to create a DAB file from an existing `docker-compose.yml` file
- Must run in the folder where the compose file is located
- Services defined in the compose file must use pre-built images
  - Cannot use `build` instruction
  - Use `image` instruction to pull existing image from registry
- Default file name is `<name of project folder>.dab`
- **Note:** volumes are currently not supported

docker

Ex 9.17 – Create our Bundle

EXERCISE

1. Change directory in the `orchestration-workshop/dockercoins` folder
2. Open `docker-compose.yml` in a text editor
3. Change the `build` instruction on each service to an `image` instruction and use the pre built image from your Docker Hub account for that respective service
4. Build the Distributed Application Bundle
   `$ docker-compose bundle`
5. Confirm that you can now see a `inventoryservice.dab` file in your dockercoins folder

Deploying a DAB

- Use the `docker deploy` command and specify the DAB file
  - Do not specify the .dab extension, just the filename.

```
ubuntu@node-0:~/orchestration-workshop/dockercoins$ docker-compose bundle
WARNING: Unsupported key 'volumes' in services.webui - ignoring
Wrote bundle to dockercoins.dab
ubuntu@node-0:~/orchestration-workshop/dockercoins$ docker deploy dockercoins
Loading bundle from dockercoins.dab
Creating network dockercoins_default
Creating service dockercoins_hasher
Creating service dockercoins_redis
Creating service dockercoins_rng
Creating service dockercoins_webui
Creating service dockercoins_worker
```

Viewing your stack

- Use `docker stack` command
  - `docker stack ps <bundle name>` - check the tasks for all services
  - `docker stack config <bundle name>` - check the DAB configuration
- Can also use the `docker service ps` command to check on individual services

Deployment caveats

- At the time of release for this training course, port mappings defined in the Compose file are not reflected in the DAB bundle.
- Services with ports exposed will be automatically mapped to port numbers starting from 30000
- You can manually change the port mapping by updating the service. For example:

```
docker service update <service name> --publish-add
8080:80
```

Deleting the stack

- Run `docker stack rm <DAB name>`
- Removes all the services associated with the bundle
- More convenient than manually removing services with the `docker service rm` command
- Must be run in the same folder where the bundle file is located

docker

Ex 9.18 – Deploy our stack

EXERCISE

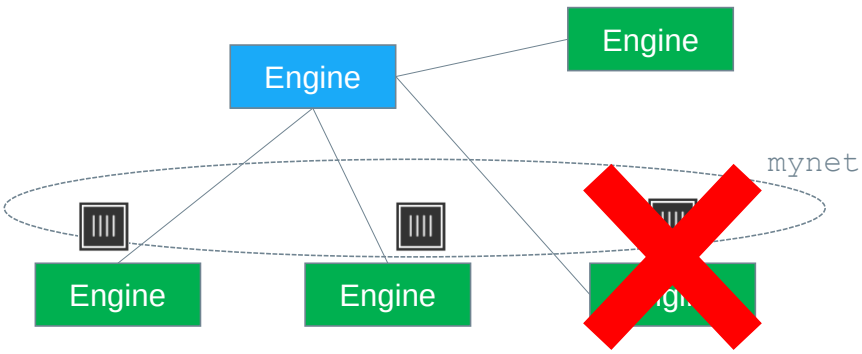
1. Make sure you are in your `dockercoins` folder
2. Deploy our DAB
   ```
   $ docker deploy inventoryservice
   ```
3. Run `docker service ls` to check that our 5 services have been created
4. Check the individual stack tasks
   ```
   $ docker stack ps myapplication
   ```
5. Delete the stack
   ```
   $ docker stack rm myapplication
   ```


docker

Swarm mode – part 5: Additional Swarm operations and further information

docker

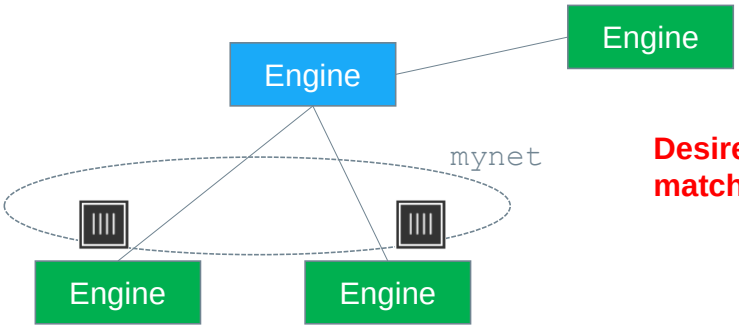
Node failure




```
$ docker service create --replicas 3 --name myapp
--network mynet --publish 80:80 myapp:1.0
```


docker

Node failure (cont'd)

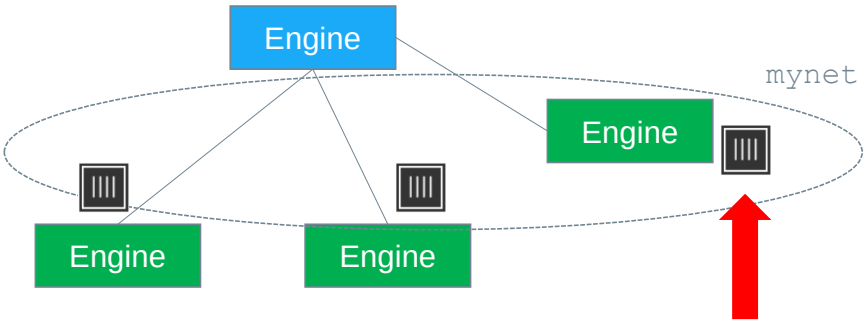


Desired state no longer matches actual state

```
$ docker service create --replicas 3 --name myapp
--network mynet --publish 80:80 myapp:1.0
```


docker

Node failure (cont'd)



Swarm will schedule a new task in order to create the new container so that we once again have 3 replicas

docker

Ex 9.19 – Simulate node failure

EXERCISE

Using your master node

1. Make sure you have no services running in your swarm
2. Create a new service called nginx, specifying 4 replicas and using the nginx image
   ```
   $ docker service create --replicas 4 --name nginx nginx
   ```
3. Run `watch "docker service ps nginx"` and observe the output. You should see 1 task on each node. Keep this terminal here.
4. Open a new terminal and SSH into node-3

docker

Ex 9.19 – (cont'd)

EXERCISE

Using the node-3 terminal

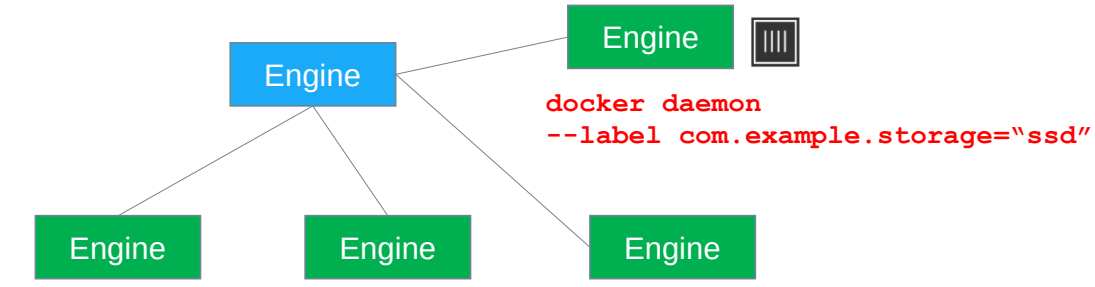
5. Reboot the node
   ```
   $ sudo reboot now
   ```
6. Switch back to your master node terminal

Using master node

7. Observe the watch command output on the terminal. You should be able to see the desired state of the node-3 task change to shutdown. A new task will be will be created to replace it. The task will be scheduled onto a different node

docker

Node constraints




```
$ docker service create --replicas 3 --name myapp
--network mynet --publish 80:80 \
--constraint com.example.storage="ssd" myapp:1.0
```


docker

Service endpoint modes

- Services can be published using two modes:
  - Virtual IP (VIP)
  - DNS Round Robin (DNSRR)
- With VIP, you get a virtual IP for the service, and an load balancer based on IPVS
- With DNSRR, you get the former behavior (from Engine 1.11), where resolving the service yields the IP addresses of all the containers for this service
- Default mode is VIP
- You change this with `docker service create --endpoint-mode [VIP|DNSRR]`

docker

Remove a node

- Nodes can be removed from a Swarm with
  `docker node rm <node hostname>`
- Should shutdown the node first before removing from the Swarm
- Manager nodes must be demoted to workers before being removed

docker

Classic Swarm

- Old Docker Swarm project at https://github.com/docker/swarm
- Serves standard Docker API
- Will slowly be replaced by new Engine 1.12 Swarm mode based on Swarmkit
- No plans to deprecate at this stage

docker

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