

Docker Orchestration Workshop

A brief introduction

- This was initially written to support in-person, instructor-led workshops and tutorials
- You can also follow along on your own, at your own pace
- We included as much information as possible in these slides
- We recommend having a mentor to help you ...
- ... Or be comfortable spending some time reading the Docker [documentation](#) ...
- ... And looking for answers in the [Docker forums](#), [StackOverflow](#), and other outlets

Hands on, you shall practice

- Nobody ever became a Jedi by spending their lives reading Wookiepedia
- Likewise, it will take more than merely *reading* these slides to make you an expert
- These slides include *tons* of exercises
- They assume that you have access to a cluster of Docker nodes
- If you are attending a workshop or tutorial:
you will be given specific instructions to access your cluster
- If you are doing this on your own:
you can use [Play-With-Docker](#) and read [these instructions](#) for extra details

Pre-requirements

- Computer with internet connection and a web browser
- For instructor-led workshops: an SSH client to connect to remote machines
 - on Linux, OS X, FreeBSD... you are probably all set
 - on Windows, get [putty](#), Microsoft [Win32 OpenSSH](#), [Git BASH](#), or [MobaXterm](#)
- For self-paced learning: SSH is not necessary if you use [Play-With-Docker](#)
- Some Docker knowledge
(but that's OK if you're not a Docker expert!)



Extra details

- This slide should have a little magnifying glass in the top right corner
(If it doesn't, it's because CSS is hard — Jérôme is only a backend person, alas)
- Slides with that magnifying glass indicate slides providing extra details
- Feel free to skip them if you're in a hurry!

Hands-on sections

- The whole workshop is hands-on
- We will see Docker in action
- You are invited to reproduce all the demos
- All hands-on sections are clearly identified, like the gray rectangle below

Exercise

- This is the stuff you're supposed to do!
- Go to [container.training](#) to view these slides
- Join the [chat room](#)

How to get your own Docker nodes?

- Use [Play-With-Docker!](#)

How to get your own Docker nodes?

- Use [Play-With-Docker](#)!
- Main differences:
 - you don't need to SSH to the machines
(just click on the node that you want to control in the left tab bar)
 - Play-With-Docker automagically detects exposed ports
(and displays them as little badges with port numbers, above the terminal)
 - You can access HTTP services by clicking on the port numbers
 - exposing TCP services requires something like [ngrok](#) or [supergrok](#)

Using Play-With-Docker

- Open a new browser tab to www.play-with-docker.com
- Confirm that you're not a robot
- Click on "ADD NEW INSTANCE": congratulations, you have your first Docker node!
- When you will need more nodes, just click on "ADD NEW INSTANCE" again
- Note the countdown in the corner; when it expires, your instances are destroyed
- If you give your URL to somebody else, they can access your nodes too
(You can use that for pair programming, or to get help from a mentor)
- Loving it? Not loving it? Tell it to the wonderful authors, [@marcosnils](#) & [@xetorthio](#)!

We will (mostly) interact with node1 only

- Unless instructed, **all commands must be run from the first VM, node1**
- We will only checkout/copy the code on node1
- When we will use the other nodes, we will do it mostly through the Docker API
- We will log into other nodes only for initial setup and a few "out of band" operations (checking internal logs, debugging...)

Terminals

Once in a while, the instructions will say:
"Open a new terminal."

There are multiple ways to do this:

- create a new window or tab on your machine, and SSH into the VM;
- use screen or tmux on the VM and open a new window from there.

You are welcome to use the method that you feel the most comfortable with.

Tmux cheatsheet

- Ctrl-b c → creates a new window
- Ctrl-b n → go to next window
- Ctrl-b p → go to previous window
- Ctrl-b " → split window top/bottom
- Ctrl-b % → split window left/right
- Ctrl-b Alt-1 → rearrange windows in columns
- Ctrl-b Alt-2 → rearrange windows in rows
- Ctrl-b arrows → navigate to other windows
- Ctrl-b d → detach session
- tmux attach → reattach to session

Brand new versions!

- Engine 17.05
- Compose 1.12
- Machine 0.11

Exercise

- Check all installed versions:

```
docker version  
docker-compose -v  
docker-machine -v
```

Wait, what, 17.05 ?!?

Wait, what, 17.05 ?!?

- Docker inc. [recently announced](#) Docker Enterprise Edition

Wait, what, 17.05 ?!?

- Docker inc. [recently announced](#) Docker Enterprise Edition
- Docker 1.13 = Docker 17.03 (year.month, like Ubuntu)
- Every month, there is a new "edge" release (with new features)
- Every quarter, there is a new "stable" release
- Docker CE releases are maintained 4+ months
- Docker EE releases are maintained 12+ months



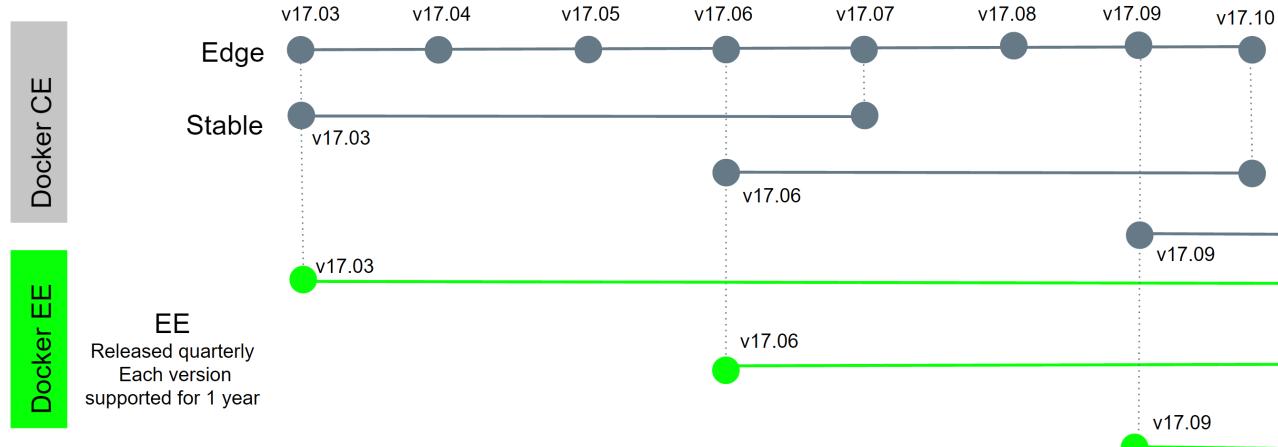
Docker CE vs Docker EE

- Docker EE:
 - \$\$\$
 - certification for select distros, clouds, and plugins
 - advanced management features (fine-grained access control, security scanning...)
- Docker CE:
 - free
 - available through Docker Mac, Docker Windows, and major Linux distros
 - perfect for individuals and small organizations



Why?

- More readable for enterprise users
 - (i.e. the very nice folks who are kind enough to pay us big \$\$\$ for our stuff)
- No impact for the community
 - (beyond CE/EE suffix and version numbering change)
- Both trains leverage the same open source components
 - (containerd, libcontainer, SwarmKit...)
- More predictable release schedule (see next slide)



All right!
We're all set.
Let's do this.

Part 1

Our sample application

- Visit the GitHub repository with all the materials of this workshop:
<https://github.com/jpetazzo/orchestration-workshop>
- The application is in the `dockercoins` subdirectory
- Let's look at the general layout of the source code:

there is a Compose file `docker-compose.yml` ...

... and 4 other services, each in its own directory:

- `rng` = web service generating random bytes
- `hasher` = web service computing hash of POSTed data
- `worker` = background process using `rng` and `hasher`
- `webui` = web interface to watch progress



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

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 did require "links" sections
- Our code can connect to services using their short name
(instead of e.g. IP address or FQDN)

Example in worker/worker.py

```
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":
```

What's this application?



(DockerCoins 2016 logo courtesy of [@XtlCnslt](#) and [@ndelooft](#). Thanks!)

What's this application?

- It is a DockerCoin miner!
- No, you can't buy coffee with DockerCoins
- How DockerCoins works:
 - `worker` asks to `rng` to give it random bytes
 - `worker` feeds those random bytes into `hasher`
 - each hash starting with `0` is a DockerCoin
 - DockerCoins are stored in `redis`
 - `redis` is also updated every second to track speed
 - you can see the progress with the `webui`

Getting the application source code

- We will clone the GitHub repository
- The repository also contains scripts and tools that we will use through the workshop

Exercise

- Clone the repository on `node1`:

```
git clone git://github.com/jpetazzo/orchestration-workshop
```

(You can also fork the repository on GitHub and clone your fork if you prefer that.)

Running the application

Without further ado, let's start our application.

Exercise

- Go to the `dockercoins` directory, in the cloned repo:

```
cd ~/orchestration-workshop/dockercoins
```

- Use Compose to build and run all containers:

```
docker-compose up
```

Compose tells Docker to build all container images (pulling the corresponding base images), then starts all containers, and displays aggregated logs.

Lots of logs

- The application continuously generates logs
- We can see the `worker` service making requests to `rng` and `hasher`
- Let's put that in the background

 Exercise

- Stop the application by hitting `^C`
- `^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)

Restarting in the background

- Many flags and commands of Compose are modeled after those of `docker`

Exercise

- Start the app in the background with the `-d` option:

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

- Check that our app is running with the `ps` command:

```
docker-compose ps
```

`docker-compose ps` also shows the ports exposed by the application.



Viewing logs

- The `docker-compose logs` command works like `docker logs`

Exercise

- View all logs since container creation and exit when done:

```
docker-compose logs
```

- Stream container logs, starting at the last 10 lines for each container:

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

Tip: use `^S` and `^Q` to pause/resume log output.



Upgrading from Compose 1.6

⚠ The `logs` command has changed between Compose 1.6 and 1.7!

- Up to 1.6
 - `docker-compose logs` is the equivalent of `logs --follow`
 - `docker-compose logs` must be restarted if containers are added
- Since 1.7
 - `--follow` must be specified explicitly
 - new containers are automatically picked up by `docker-compose logs`

Connecting to the web UI

- The `webui` container exposes a web dashboard; let's view it

Exercise

- With a web browser, connect to `node1` on port 8000
- Remember: the `nodeX` aliases are valid only on the nodes themselves
- In your browser, you need to enter the IP address of your node

You should see a speed of approximately 4 hashes/second.

More precisely: 4 hashes/second, with regular dips down to zero.
This is because Jérôme is incapable of writing good frontend code.
Don't ask. Seriously, don't ask. This is embarrassing.



Why does the speed seem irregular?

- The app actually has a constant, steady speed: 3.33 hashes/second (which corresponds to 1 hash every 0.3 seconds, for *reasons*)
- The worker doesn't update the counter after every loop, but up to once per second
- The speed is computed by the browser, checking the counter about once per second
- Between two consecutive updates, the counter will increase either by 4, or by 0
- The perceived speed will therefore be 4 - 4 - 4 - 0 - 4 - 4 - etc.

We told you to not ask!!!

Scaling up the application

- Our goal is to make that performance graph go up (without changing a line of code!)

Scaling up the application

- Our goal is to make that performance graph go up (without changing a line of code!)
- Before trying to scale the application, we'll figure out if we need more resources
(CPU, RAM...)
- For that, we will use good old UNIX tools on our Docker node

Looking at resource usage

- Let's look at CPU, memory, and I/O usage

Exercise

- run `top` to see CPU and memory usage (you should see idle cycles)
- run `vmstat 3` to see I/O usage (si/so/bi/bo)
(the 4 numbers should be almost zero, except bo for logging)

We have available resources.

- Why?
- How can we use them?

Scaling workers on a single node

- Docker Compose supports scaling
- Let's scale `worker` and see what happens!

Exercise

- Start one more `worker` container:

```
docker-compose scale worker=2
```

- Look at the performance graph (it should show a x2 improvement)
- Look at the aggregated logs of our containers (`worker_2` should show up)
- Look at the impact on CPU load with e.g. top (it should be negligible)

Adding more workers

- Great, let's add more workers and call it a day, then!

Exercise

- Start eight more `worker` containers:

```
docker-compose scale worker=10
```

- Look at the performance graph: does it show a x10 improvement?
- Look at the aggregated logs of our containers
- Look at the impact on CPU load and memory usage

Identifying bottlenecks

- You should have seen a 3x speed bump (not 10x)
- Adding workers didn't result in linear improvement
- *Something else* is slowing us down

Identifying bottlenecks

- You should have seen a 3x speed bump (not 10x)
- Adding workers didn't result in linear improvement
- *Something else* is slowing us down
- ... But what?

Identifying bottlenecks

- You should have seen a 3x speed bump (not 10x)
- Adding workers didn't result in linear improvement
- *Something else* is slowing us down
- ... But what?
- The code doesn't have instrumentation
- Let's use state-of-the-art HTTP performance analysis!
(i.e. good old tools like `ab`, `httping`...)

Accessing internal services

- `rng` and `hasher` are exposed on ports 8001 and 8002
- This is declared in the Compose file:

```
...
rng:
  build: rng
  ports:
    - "8001:80"

hasher:
  build: hasher
  ports:
    - "8002:80"
...
```

Measuring latency under load

We will use `httping`.

Exercise

- Check the latency of `rng`:

```
httping -c 10 localhost:8001
```

- Check the latency of `hasher`:

```
httping -c 10 localhost:8002
```

`rng` has a much higher latency than `hasher`.

Let's draw hasty conclusions

- The bottleneck seems to be `rng`
- *What if* we don't have enough entropy and can't generate enough random numbers?
- We need to scale out the `rng` service on multiple machines!

Note: this is a fiction! We have enough entropy. But we need a pretext to scale out.

(In fact, the code of `rng` uses `/dev/urandom`, which never runs out of entropy...
...and is just as good as `/dev/random`.)

Clean up

- Before moving on, let's remove those containers

 Exercise

- Tell Compose to remove everything:

```
docker-compose down
```

Scaling out

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/swarmd/swarmctl → libcontainer/containerd/container-ctr

SwarmKit features

- Highly-available, distributed store based on [Raft](#)
(avoids depending on an external store: easier to deploy; higher performance)
- Dynamic reconfiguration of Raft without interrupting cluster operations
- Services managed with a *declarative API*
(implementing *desired state* and *reconciliation loop*)
- Integration with overlay networks and load balancing
- Strong emphasis on security:
 - automatic TLS keying and signing; automatic cert rotation
 - full encryption of the data plane; automatic key rotation
 - least privilege architecture (single-node compromise ≠ cluster compromise)
 - on-disk encryption with optional passphrase



Where is the key/value store?

- Many orchestration systems use a key/value store backed by a consensus algorithm (k8s→etcd→Raft, mesos→zookeeper→ZAB, etc.)
- SwarmKit implements the Raft algorithm directly
(Nomad is similar; thanks [@cbednarski](#), [@diptanu](#) and others for pointing it out!)
- Analogy courtesy of [@aluzzardi](#):

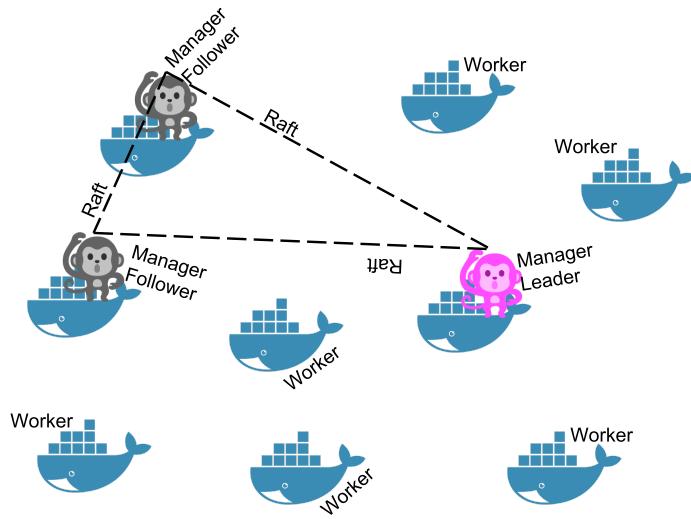
It's like B-Trees and RDBMS. They are different layers, often associated. But you don't need to bring up a full SQL server when all you need is to index some data.

- As a result, the orchestrator has direct access to the data
(the main copy of the data is stored in the orchestrator's memory)
- Simpler, easier to deploy and operate; also faster

SwarmKit concepts (1/2)

- A *cluster* will be at least one *node* (preferably more)
- A *node* can be a *manager* or a *worker*
- A *manager* actively takes part in the Raft consensus, and keeps the Raft log
- You can talk to a *manager* using the SwarmKit API
- One *manager* is elected as the *leader*; other managers merely forward requests to it
- The *workers* get their instructions from the *managers*
- Both *workers* and *managers* can run containers

Illustration



SwarmKit concepts (2/2)

- The *managers* expose the SwarmKit API
- Using the API, you can indicate that you want to run a *service*
- A *service* is specified by its *desired state*: which image, how many instances...
- The *leader* uses different subsystems to break down services into *tasks*: orchestrator, scheduler, allocator, dispatcher
- A *task* corresponds to a specific container, assigned to a specific *node*
- *Nodes* know which *tasks* should be running, and will start or stop containers accordingly (through the Docker Engine API)

You can refer to the [NOMENCLATURE](#) in the SwarmKit repo for more details.

Swarm Mode

- Since version 1.12, Docker Engine embeds SwarmKit
- All the SwarmKit features are "asleep" until you enable "Swarm Mode"
- Examples of Swarm Mode commands:
 - `docker swarm` (enable Swarm mode; join a Swarm; adjust cluster parameters)
 - `docker node` (view nodes; promote/demote managers; manage nodes)
 - `docker service` (create and manage services)

You need to enable Swarm mode to use the new stuff

- By default, all this new code is inactive
- Swarm Mode can be enabled, "unlocking" SwarmKit functions (services, out-of-the-box overlay networks, etc.)

Exercise

- Try a Swarm-specific command:

```
docker node ls
```

You need to enable Swarm mode to use the new stuff

- By default, all this new code is inactive
- Swarm Mode can be enabled, "unlocking" SwarmKit functions (services, out-of-the-box overlay networks, etc.)

Exercise

- Try a Swarm-specific command:

```
docker node ls
```

You will get an error message:

```
Error response from daemon: This node is not a swarm manager. [...]
```

Creating our first Swarm

- 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 disjoint clusters.

Exercise

- Create our cluster from node1:

```
docker swarm init
```

Creating our first Swarm

- 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 disjoint clusters.

Exercise

- Create our cluster from node1:

```
docker swarm init
```

If Docker tells you that it `could not choose an IP address to advertise`, see next slide!

IP address to advertise

- When running in Swarm mode, each node *advertises* its address to the others
(i.e. it tells them "*you can contact me on 10.1.2.3:2377*")
- If the node has only one IP address (other than 127.0.0.1), it is used automatically
- If the node has multiple IP addresses, you **must** specify which one to use
(Docker refuses to pick one randomly)
- You can specify an IP address or an interface name
(in the latter case, Docker will read the IP address of the interface and use it)
- You can also specify a port number
(otherwise, the default port 2377 will be used)

Which IP address should be advertised?

- If your nodes have only one IP address, it's safe to let autodetection do the job

(Except if your instances have different private and public addresses, e.g. on EC2, and you are building a Swarm involving nodes inside and outside the private network: then you should advertise the public address.)

- If your nodes have multiple IP addresses, pick an address which is reachable by *every other node* of the Swarm
- If you are using [play-with-docker](#), use the IP address shown next to the node name

(This is the address of your node on your private internal overlay network. The other address that you might see is the address of your node on the `docker_gwbridge` network, which is used for outbound traffic.)

Examples:

```
docker swarm init --advertise-addr 10.0.9.2
docker swarm init --advertise-addr eth0:7777
```

Token generation

- In the output of `docker swarm init`, we have a message confirming that our node is now the (single) manager:

```
Swarm initialized: current node (8jud...) is now a manager.
```

- Docker generated two security tokens (like passphrases or passwords) for our cluster
- The CLI shows us the command to use on other nodes to add them to the cluster using the "worker" security token:

```
To add a worker to this swarm, run the following command:  
  docker swarm join \  
    --token SWMTKN-1-59fl4ak4nqjmao1ofttrc4eprhrola2l87... \  
    172.31.4.182:2377
```



Checking that Swarm mode is enabled

Exercise

- Run the traditional docker info command:

```
docker info
```

The output should include:

```
Swarm: active
NodeID: 8jud7o8dax3zxbags3f8yox4b
Is Manager: true
ClusterID: 2vcw2oa9rjps3a24m91xhv0c
...
```

Running our first Swarm mode command

- Let's retry the exact same command as earlier

 Exercise

- List the nodes (well, the only node) of our cluster:

```
docker node ls
```

The output should look like the following:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
8jud...ox4b *	node1	Ready	Active	Leader

Adding nodes to the Swarm

- A cluster with one node is not a lot of fun
- Let's add `node2`!
- We need the token that was shown earlier

Adding nodes to the Swarm

- A cluster with one node is not a lot of fun
- Let's add `node2`!
- We need the token that was shown earlier
- You wrote it down, right?

Adding nodes to the Swarm

- A cluster with one node is not a lot of fun
- Let's add `node2`!
- We need the token that was shown earlier
- You wrote it down, right?
- Don't panic, we can easily see it again

Adding nodes to the Swarm

Exercise

- Show the token again:

```
docker swarm join-token worker
```

- Switch to `node2`
- Copy-paste the `docker swarm join ...` command
(that was displayed just before)



Check that the node was added correctly

- Stay on `node2` for now!

Exercise

- We can still use `docker info` to verify that the node is part of the Swarm:

```
docker info | grep ^Swarm
```

- However, Swarm commands will not work; try, for instance:

```
docker node ls
```

- This is because the node that we added is currently a *worker*
- Only *managers* can accept Swarm-specific commands

View our two-node cluster

- Let's go back to `node1` and see what our cluster looks like

Exercise

- Switch back to `node1`
- View the cluster from `node1`, which is a manager:

```
docker node ls
```

The output should be similar to the following:

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
8jud...ox4b *	node1	Ready	Active	Leader
ehb0...4fvx	node2	Ready	Active	

Adding nodes using the Docker API

- We don't have to SSH into the other nodes, we can use the Docker API
- If you are using Play-With-Docker:
 - the nodes expose the Docker API over port 2375/tcp, without authentication
 - we will connect by setting the `DOCKER_HOST` environment variable
- Otherwise:
 - the nodes expose the Docker API over port 2376/tcp, with TLS mutual authentication
 - we will use Docker Machine to set the correct environment variables
(the nodes have been suitably pre-configured to be controlled through `node1`)

Docker Machine

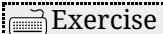
- Docker Machine has two primary uses:
 - provisioning cloud instances running the Docker Engine
 - managing local Docker VMs within e.g. VirtualBox
- Docker Machine is purely optional
- It makes it easy to create, upgrade, manage... Docker hosts:
 - on your favorite cloud provider
 - locally (e.g. to test clustering, or different versions)
 - across different cloud providers

If you're using Play-With-Docker ...

- You won't need to use Docker Machine
- Instead, to "talk" to another node, we'll just set `DOCKER_HOST`
- You can skip the exercises telling you to do things with Docker Machine!

Docker Machine basic usage

- We will learn two commands:
 - `docker-machine ls` (list existing hosts)
 - `docker-machine env` (switch to a specific host)



- List configured hosts:

```
docker-machine ls
```

You should see your 5 nodes.

Using Docker Machine to communicate with a node

- To select a node, use `eval $(docker-machine env nodeX)`
- This sets a number of environment variables
- To unset these variables, use `eval $(docker-machine env -u)`

Exercise

- View the variables used by Docker Machine:

```
docker-machine env node3
```

(This shows which variables *would* be set by Docker Machine; but it doesn't change them.)

Getting the token

- First, let's store the join token in a variable
- This must be done from a manager

Exercise

- Make sure we talk to the local node, or `node1`:

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

- Get the join token:

```
TOKEN=$(docker swarm join-token -q worker)
```

Change the node targeted by the Docker CLI

- We need to set the right environment variables to communicate with `node3`

Exercise

- If you're using Play-With-Docker:

```
export DOCKER_HOST=tcp://node3:2375
```

- Otherwise, use Docker Machine:

```
eval $(docker-machine env node3)
```

Checking which node we're talking to

- Let's use the Docker API to ask "who are you?" to the remote node

Exercise

- Extract the node name from the output of `docker info`:

```
docker info | grep ^Name
```

This should tell us that we are talking to `node3`.

Note: it can be useful to use a [custom shell prompt](#) reflecting the `DOCKER_HOST` variable.

Adding a node through the Docker API

- We are going to use the same `docker swarm join` command as before

 Exercise

- Add `node3` to the Swarm:

```
docker swarm join --token $TOKEN node1:2377
```

Going back to the local node

- We need to revert the environment variable(s) that we had set previously

Exercise

- If you're using Play-With-Docker, just clear `DOCKER_HOST`:

```
unset DOCKER_HOST
```

- Otherwise, use Docker Machine to reset all the relevant variables:

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

From that point, we are communicating with `node1` again.

Checking the composition of our cluster

- Now that we're talking to `node1` again, we can use management commands

 Exercise

- Check that the node is here:

```
docker node ls
```

Under the hood: docker swarm init

When we do `docker swarm init`:

- a keypair is created for the root CA of our Swarm
- a keypair is created for the first node
- a certificate is issued for this node
- the join tokens are created

Under the hood: join tokens

There is one token to *join as a worker*, and another to *join as a manager*.

The join tokens have two parts:

- a secret key (preventing unauthorized nodes from joining)
- a fingerprint of the root CA certificate (preventing MITM attacks)

If a token is compromised, it can be rotated instantly with:

```
docker swarm join-token --rotate <worker|manager>
```

Under the hood: docker swarm join

When a node joins the Swarm:

- it is issued its own keypair, signed by the root CA
- if the node is a manager:
 - it joins the Raft consensus
 - it connects to the current leader
 - it accepts connections from worker nodes
- if the node is a worker:
 - it connects to one of the managers (leader or follower)

Under the hood: cluster communication

- The *control plane* is encrypted with AES-GCM; keys are rotated every 12 hours
- Authentication is done with mutual TLS; certificates are rotated every 90 days
(`docker swarm update` allows to change this delay or to use an external CA)
- The *data plane* (communication between containers) is not encrypted by default
(but this can be activated on a by-network basis, using IPSEC, leveraging hardware crypto if available)

Under the hood: I want to know more!

Revisit SwarmKit concepts:

- Docker 1.12 Swarm Mode Deep Dive Part 1: Topology ([video](#))
- Docker 1.12 Swarm Mode Deep Dive Part 2: Orchestration ([video](#))

Some presentations from the Docker Distributed Systems Summit in Berlin:

- Heart of the SwarmKit: Topology Management ([slides](#))
- Heart of the SwarmKit: Store, Topology & Object Model ([slides](#)) ([video](#))

Adding more manager nodes

- Right now, we have only one manager (node1)
- If we lose it, we lose quorum - and that's *very bad!*
- Containers running on other nodes will be fine ...
- But we won't be able to get or set anything related to the cluster
- If the manager is permanently gone, we will have to do a manual repair!
- Nobody wants to do that ... so let's make our cluster highly available

Adding more managers

With Play-With-Docker:

```
TOKEN=$(docker swarm join-token -q manager)
for N in $(seq 4 5); do
    export DOCKER_HOST=tcp://node$N:2375
    docker swarm join --token $TOKEN node1:2377
done
unset DOCKER_HOST
```

Adding more managers

With Docker Machine:

```
TOKEN=$(docker swarm join-token -q manager)
for N in $(seq 4 5); do
    eval $(docker-machine env node$N)
    docker swarm join --token $TOKEN node1:2377
done
eval $(docker-machine env -u)
```

You can control the Swarm from any manager node

Exercise

- Try the following command on a few different nodes:

```
docker node ls
```

On manager nodes:

you will see the list of nodes, with a `*` denoting the node you're talking to.

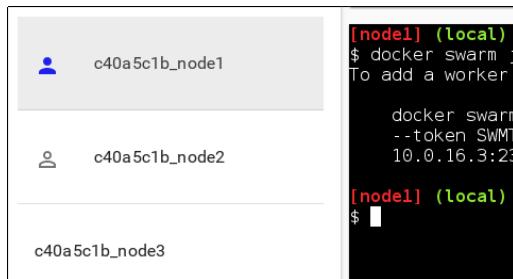
On non-manager nodes:

you will get an error message telling you that the node is not a manager.

As we saw earlier, you can only control the Swarm through a manager node.

Play-With-Docker node status icon

- If you're using Play-With-Docker, you get node status icons
- Node status icons are displayed left of the node name
 - No icon = no Swarm mode detected
 - Solid blue icon = Swarm manager detected
 - Blue outline icon = Swarm worker detected



```
[node1] (local)
$ docker swarm join --token SWMT-10.0.16.3:23
[...]
[node1] (local)
$
```

Dynamically changing the role of a node

- We can change the role of a node on the fly:

```
docker node promote XXX → make XXX a manager  
docker node demote XXX → make XXX a worker
```

Exercise

- See the current list of nodes:

```
docker node ls
```

- Promote any worker node to be a manager:

```
docker node promote <node_name_or_id>
```

How many managers do we need?

- $2N+1$ nodes can (and will) tolerate N failures
(you can have an even number of managers, but there is no point)

How many managers do we need?

- $2N+1$ nodes can (and will) tolerate N failures
(you can have an even number of managers, but there is no point)
- 1 manager = no failure
- 3 managers = 1 failure
- 5 managers = 2 failures (or 1 failure during 1 maintenance)
- 7 managers and more = now you might be overdoing it a little bit

Why not have *all* nodes be managers?

- Intuitively, it's harder to reach consensus in larger groups
- With Raft, writes have to go to (and be acknowledged by) all nodes
- More nodes = more network traffic
- Bigger network = more latency

What would McGyver do?

- If some of your machines are more than 10ms away from each other, try to break them down in multiple clusters (keeping internal latency low)
- Groups of up to 9 nodes: all of them are managers
- Groups of 10 nodes and up: pick 5 "stable" nodes to be managers
(Cloud pro-tip: use separate auto-scaling groups for managers and workers)
- Groups of more than 100 nodes: watch your managers' CPU and RAM
- Groups of more than 1000 nodes:
 - if you can afford to have fast, stable managers, add more of them
 - otherwise, break down your nodes in multiple clusters

What's the upper limit?

- We don't know!
- Internal testing at Docker Inc.: 1000-10000 nodes is fine
 - deployed to a single cloud region
 - one of the main take-aways was "*you're gonna need a bigger manager*"
- Testing by the community: [4700 heterogenous nodes all over the 'net](#)
 - it just works
 - more nodes require more CPU; more containers require more RAM
 - scheduling of large jobs (70000 containers) is slow, though (working on it!)

Running our first Swarm service

- How do we run services? Simplified version:

```
docker run → docker service create
```

Exercise

- Create a service featuring an Alpine container pinging Google resolvers:

```
docker service create alpine ping 8.8.8.8
```

- Check the result:

```
docker service ps <serviceID>
```

--detach for service creation

(New in Docker Engine 17.05)

If you are running Docker 17.05, you will see the following message:

Since `--detach=false` was not specified, tasks will be created in the background.
In a future release, `--detach=false` will become the default.

Let's ignore it for now; but we'll come back to it in just a few minutes!

Checking service logs

(New in Docker Engine 17.05)

- Just like `docker logs` shows the output of a specific local container ...
- ... `docker service logs` shows the output of all the containers of a specific service

Exercise

- Check the output of our ping command:

```
docker service logs <serviceID>
```

Flags `--follow` and `--tail` are available, as well as a few others.

Note: by default, when a container is destroyed (e.g. when scaling down), its logs are lost.

Before Docker Engine 17.05

- Docker 1.13/17.03/17.04 have `docker service logs` as an experimental feature (available only when enabling the experimental feature flag)
- We have to use `docker logs`, which only works on local containers
- We will have to connect to the node running our container (unless it was scheduled locally, of course)



Looking up where our container is running

- The `docker service ps` command told us where our container was scheduled

Exercise

- Look up the `NODE` on which the container is running:

```
docker service ps <serviceID>
```

- If you use Play-With-Docker, switch to that node's tab, or set `DOCKER_HOST`
- Otherwise, `ssh` into the node or use `$(eval docker-machine env node...)`



Viewing the logs of the container

Exercise

- See that the container is running and check its ID:

```
docker ps
```

- View its logs:

```
docker logs <containerID>
```

- Go back to `node1` afterwards

Scale our service

- Services can be scaled in a pinch with the `docker service update` command

Exercise

- Scale the service to ensure 2 copies per node:

```
docker service update <serviceID> --replicas 10
```

- Check that we have two containers on the current node:

```
docker ps
```

View deployment progress

(New in Docker Engine 17.05)

- Commands that create/update/delete services can run with `--detach=false`
- The CLI will show the status of the command, and exit once it's done working



- Scale the service to ensure 3 copies per node:

```
docker service update <serviceID> --replicas 15 --detach=false
```

Note: `--detach=false` will eventually become the default.

With older versions, you can use e.g.: `watch docker service ps <serviceID>`

Expose a service

- Services can be exposed, 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:

```
docker run -p → docker service create -p
```

- 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
(just like with `docker run -p`)

Expose ElasticSearch on its default port

Exercise

- Create an ElasticSearch service (and give it a name while we're at it):

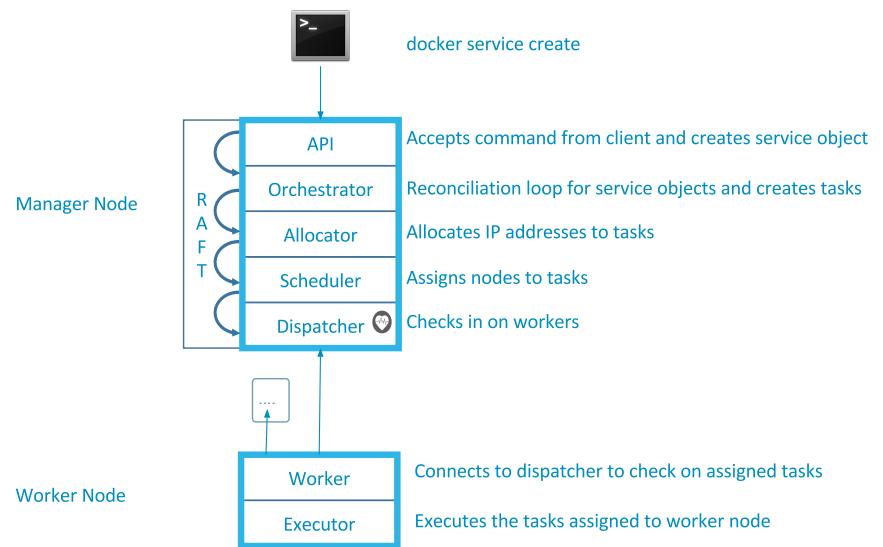
```
docker service create --name search --publish 9200:9200 --replicas 7 \
--detach=false elasticsearch:2
```

Note: don't forget the :2!

The latest version of the ElasticSearch image won't start without mandatory configuration.

Tasks lifecycle

- During the deployment, you will be able to see multiple states:
 - assigned (the task has been assigned to a specific node)
 - preparing (this mostly means "pulling the image")
 - starting
 - running
- When a task is terminated (stopped, killed...) it cannot be restarted
(A replacement task will be created)



Test our service

- We mapped port 9200 on the nodes, to port 9200 in the containers
- Let's try to reach that port!

Exercise

- Try the following command:

```
curl localhost:9200
```

(If you get `Connection refused`: congratulations, you are very fast indeed! Just try again.)

ElasticSearch serves a little JSON document with some basic information about this instance; including a randomly-generated super-hero name.

Test the load balancing

- If we repeat our `curl` command multiple times, we will see different names

Exercise

- Send 10 requests, and see which instances serve them:

```
for N in $(seq 1 10); do
    curl -s localhost:9200 | jq .name
done
```

Note: if you don't have `jq` on your Play-With-Docker instance, just install it:

```
apk add --no-cache jq
```

Load balancing results

Traffic is handled by our clusters [TCP routing mesh](#).

Each request is served by one of the 7 instances, in rotation.

Note: if you try to access the service from your browser, you will probably see the same instance name over and over, because your browser (unlike curl) will try to re-use the same connection.

Under the hood of the TCP routing mesh

- Load balancing is done by IPVS
- IPVS is a high-performance, in-kernel load balancer
- It's been around for a long time (merged in the kernel since 2.4)
- Each node runs a local load balancer

(Allowing connections to be routed directly to the destination, without extra hops)

Managing inbound traffic

There are many ways to deal with inbound traffic on a Swarm cluster.

- Put all (or a subset) of your nodes in a DNS A record
- Assign your nodes (or a subset) to an ELB
- Use a virtual IP and make sure that it is assigned to an "alive" node
- etc.

Managing HTTP traffic

- The TCP routing mesh doesn't parse HTTP headers
- If you want to place multiple HTTP services on port 80, you need something more
- You can setup NGINX or HAProxy on port 80 to do the virtual host switching
- Docker Universal Control Plane provides its own [HTTP routing mesh](#)
 - add a specific label starting with `com.docker.ucp.mesh.http` to your services
 - labels are detected automatically and dynamically update the configuration

You should use labels

- Labels are a great way to attach arbitrary information to services
- Examples:
 - HTTP vhost of a web app or web service
 - backup schedule for a stateful service
 - owner of a service (for billing, paging...)
 - etc.

Visualize container placement

- Let's leverage the Docker API!

Exercise

- Get the source code of this simple-yet-beautiful visualization app:

```
cd ~  
git clone git://github.com/dockersamples/docker-swarm-visualizer
```

- Build and run the Swarm visualizer:

```
cd docker-swarm-visualizer  
docker-compose up -d
```

Connect to the visualization webapp

- It runs a web server on port 8080

 Exercise

- Point your browser to port 8080 of your node1's public ip
(If you use Play-With-Docker, click on the (8080) badge)

- The webapp updates the display automatically (you don't need to reload the page)
- It only shows Swarm services (not standalone containers)
- It shows when nodes go down
- It has some glitches (it's not Carrier-Grade Enterprise-Compliant ISO-9001 software)

Why This Is More Important Than You Think

- The visualizer accesses the Docker API *from within a container*
- This is a common pattern: run container management tools *in containers*
- Instead of viewing your cluster, this could take care of logging, metrics, autoscaling ...
- We can run it within a service, too! We won't do it, but the command would look like:

```
docker service create \  
  --mount source=/var/run/docker.sock,type=bind,target=/var/run/docker.sock \  
  --name viz --constraint node.role==manager ...
```

Credits: the visualization code was written by [Francisco Miranda](#).
[Mano Marks](#) adapted it to Swarm and maintains it.

Terminate our services

- Before moving on, we will remove those services
- `docker service rm` can accept multiple services names or IDs
- `docker service ls` can accept the `-q` flag
- A Shell snippet a day keeps the cruft away

Exercise

- Remove all services with this one liner:

```
docker service ls -q | xargs docker service rm
```

Our app on Swarm

What's on the menu?

In this part, we will:

- **build** images for our app,
- **ship** these images with a registry,
- **run** services using these images.

Why do we need to ship our images?

- When we do `docker-compose up`, images are built for our services
- These images are present only on the local node
- We need these 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



Build, ship, and run, for a single service

If we had only one service (built from a `Dockerfile` in the current directory), our workflow could look like this:

```
docker build -t jpetazzo/doublerainbow:v0.1 .
docker push jpetazzo/doublerainbow:v0.1
docker service create jpetazzo/doublerainbow:v0.1
```

We just have to adapt this to our application, which has 4 services!

The plan

- Build on our local node (`node1`)
- Tag images so that they are named `localhost:5000/servicename`
- Upload them to a registry
- Create services using the images

Which registry do we want to use?

- **Docker Hub**

- hosted by Docker Inc.
- requires an account (free, no credit card needed)
- images will be public (unless you pay)
- located in AWS EC2 us-east-1

- **Docker Trusted Registry**

- self-hosted commercial product
- requires a subscription (free 30-day trial available)
- images can be public or private
- located wherever you want

- **Docker open source registry**

- self-hosted barebones repository hosting
- doesn't require anything
- doesn't come with anything either
- located wherever you want



Using Docker Hub

If we wanted to use the Docker Hub...

- We would log into the Docker Hub:

```
docker login
```

- And in the following slides, we would use our Docker Hub login (e.g. `jpetazzo`) instead of the registry address (i.e. `127.0.0.1:5000`)



Using Docker Trusted Registry

If we wanted to use DTR, we would...

- Make sure we have a Docker Hub account
- [Activate a Docker Datacenter subscription](#)
- Install DTR on our machines
- Use `dtraddress:port/user` instead of the registry address

This is out of the scope of this workshop!

Using the open source registry

- We need to run a `registry:2` container
(make sure you specify tag `:2` to run the new version!)
- It will store images and layers to the local filesystem
(but you can add a config file to use S3, Swift, etc.)
- Docker *requires* TLS when communicating with the registry
 - unless for registries on `127.0.0.0/8` (i.e. `localhost`)
 - or with the Engine flag `--insecure-registry`
- Our strategy: publish the registry container on port 5000,
so that it's available through `127.0.0.1:5000` on each node

Deploying a local registry

- We will create a single-instance service, publishing its port on the whole cluster

Exercise

- Create the registry service:

```
docker service create --name registry --publish 5000:5000 registry:2
```

- Try the following command, until it returns {"repositories":[]}:

curl 127.0.0.1:5000/v2/_catalog

(Retry a few times, it might take 10-20 seconds for the container to be started. Patience.)

Testing our local registry

- We can retag a small image, and push it to the registry

Exercise

- Make sure we have the busybox image, and retag it:

```
docker pull busybox
docker tag busybox 127.0.0.1:5000/busybox
```

- Push it:

```
docker push 127.0.0.1:5000/busybox
```

Checking what's on our local registry

- The registry API has endpoints to query what's there

Exercise

- Ensure that our busybox image is now in the local registry:

```
curl http://127.0.0.1:5000/v2/_catalog
```

The curl command should now output:

```
{"repositories":["busybox"]}
```

Build, tag, and push our application container images

- Compose has named our images `dockercoins_XXX` for each service
- We need to retag them (to `127.0.0.1:5000/XXX:v1`) and push them

Exercise

- Set `REGISTRY` and `TAG` environment variables to use our local registry
- And run this little for loop:

```
cd ~/orchestration-workshop/dockercoins
REGISTRY=127.0.0.1:5000 TAG=v1
for SERVICE in hasher rng webui worker; do
    docker tag dockercoins_${SERVICE} $REGISTRY/$SERVICE:$TAG
    docker push $REGISTRY/$SERVICE
done
```

Overlay networks

- SwarmKit integrates with overlay networks
- Networks are created with `docker network create`
- Make sure to specify that you want an *overlay* network
(otherwise you will get a local *bridge* network by default)

Exercise

- Create an overlay network for our application:

```
docker network create --driver overlay dockercoins
```

Viewing existing networks

- Let's confirm that our network was created

 Exercise

- List existing networks:

```
docker network ls
```

Can you spot the differences?

The networks `dockercoins` and `ingress` are different from the other ones.

Can you see how?

Can you spot the differences?

The networks `dockercoins` and `ingress` are different from the other ones.

Can you see how?

- They are using a different kind of ID, reflecting the fact that they are SwarmKit objects instead of "classic" Docker Engine objects.
- Their *scope* is `swarm` instead of `local`.
- They are using the overlay driver.



Caveats

In Docker 1.12, you cannot join an overlay network with `docker run --net ...`.

Starting with version 1.13, you can, if the network was created with the `--attachable` flag.

Why is that?

Placing a container on a network requires allocating an IP address for this container.

The allocation must be done by a manager node (worker nodes cannot update Raft data).

As a result, `docker run --net ...` requires collaboration with manager nodes.

It alters the code path for `docker run`, so it is allowed only under strict circumstances.

Run the application

- First, create the `redis` service; that one is using a Docker Hub image

 Exercise

- Create the `redis` service:

```
docker service create --network dockercoins --name redis redis
```

Run the other services

- Then, start the other services one by one
- We will use the images pushed previously

Exercise

- Start the other services:

```
REGISTRY=127.0.0.1:5000
TAG=v1
for SERVICE in hasher rng webui worker; do
  docker service create --network dockercoins --detach=true \
    --name $SERVICE $REGISTRY/$SERVICE:$TAG
done
```

Expose our application web UI

- We need to connect to the `webui` service, but it is not publishing any port
- Let's reconfigure it to publish a port

Exercise

- Update `webui` so that we can connect to it from outside:

```
docker service update webui --publish-add 8000:80 --detach=false
```

Note: to "de-publish" a port, you would have to specify the container port.
(i.e. in that case, `--publish-rm 80`)

What happens when we modify a service?

- Let's find out what happened to our `webui` service

 Exercise

- Look at the tasks and containers associated to `webui`:

```
docker service ps webui
```

What happens when we modify a service?

- Let's find out what happened to our `webui` service

 Exercise

- Look at the tasks and containers associated to `webui`:

```
docker service ps webui
```

The first version of the service (the one that was not exposed) has been shutdown.

It has been replaced by the new version, with port 80 accessible from outside.

(This will be discussed with more details in the section about stateful services.)

Connect to the web UI

- The web UI is now available on port 8000, *on all the nodes of the cluster*

 Exercise

- If you're using Play-With-Docker, just click on the (8000) badge
- Otherwise, point your browser to any node, on port 8000

Scaling the application

- We can change scaling parameters with `docker update` as well
- We will do the equivalent of `docker-compose scale`

Exercise

- Bring up more workers:

```
docker service update worker --replicas 10 --detach=false
```

- Check the result in the web UI

You should see the performance peaking at 10 hashes/s (like before).

Global scheduling

- We want to utilize as best as we can the entropy generators on our nodes
- We want to run exactly one `rng` instance per node
- SwarmKit has a special scheduling mode for that, let's use it
- We cannot enable/disable global scheduling on an existing service
- We have to destroy and re-create the `rng` service

Scaling the `rng` service

Exercise

- Remove the existing `rng` service:

```
docker service rm rng
```

- Re-create the `rng` service with *global scheduling*:

```
docker service create --name rng --network dockercoins --mode global \  
  --detach=false $REGISTRY/rng:$TAG
```

- Look at the result in the web UI



Why do we have to re-create the service to enable global scheduling?

- Enabling it dynamically would make rolling updates semantics very complex
- This might change in the future (after all, it was possible in 1.12 RC!)
- As of Docker Engine 17.05, other parameters requiring to `rm/create` the service are:
 - service name
 - hostname
 - network

How did we make our app "Swarm-ready"?

This app was written in June 2015. (One year before Swarm mode was released.)

What did we change to make it compatible with Swarm mode?

How did we make our app "Swarm-ready"?

This app was written in June 2015. (One year before Swarm mode was released.)

What did we change to make it compatible with Swarm mode?

Exercise

- Go to the app directory:

```
cd ~/orchestration-workshop/dockercoins
```

- See modifications in the code:

```
git log -p --since "4-JUL-2015" -- . ':!*.*.yml*' ':!*.*.html'
```

What did we change in our app since its inception?

- Compose files
- HTML file (it contains an embedded contextual tweet)
- Dockerfiles (to switch to smaller images)
- That's it!

What did we change in our app since its inception?

- Compose files
- HTML file (it contains an embedded contextual tweet)
- Dockerfiles (to switch to smaller images)
- That's it!

We didn't change a single line of code in this app since it was written.

What did we change in our app since its inception?

- Compose files
- HTML file (it contains an embedded contextual tweet)
- Dockerfiles (to switch to smaller images)
- That's it!

We didn't change a single line of code in this app since it was written.

*The images that were [built in June 2015](#) (when the app was written) can still run today ...
... in Swarm mode (distributed across a cluster, with load balancing) ...
... without any modification.*

How did we design our app in the first place?

- [Twelve-Factor App](#) principles
- Service discovery using DNS names
 - Initially implemented as "links"
 - Then "ambassadors"
 - And now "services"
- Existing apps might require more changes!

Integration with Compose

- The previous section showed us how to streamline image build and push
- We will now see how to streamline service creation
(i.e. get rid of the `for SERVICE in ...; do docker service create ...` part)

Compose file version 3

(New in Docker Engine 1.13)

- Almost identical to version 2
- Can be directly used by a Swarm cluster through `docker stack ...` commands
- Introduces a `deploy` section to pass Swarm-specific parameters
- Resource limits are moved to this `deploy` section
- See [here](#) for the complete list of changes
- Supersedes *Distributed Application Bundles*

(JSON payload describing an application; could be generated from a Compose file)

Removing everything

- Before deploying using "stacks," let's get a clean slate

 Exercise

- Remove *all* the services:

```
docker service ls -q | xargs docker service rm
```

Our first stack

We need a registry to move images around.

Before, we deployed it with the following command:

```
docker service create --publish 5000:5000 registry:2
```

Now, we are going to deploy it with the following stack file:

```
version: "3"

services:
  registry:
    image: registry:2
    ports:
      - "5000:5000"
```

Checking our stack files

- All the stack files that we will use are in the `stacks` directory

 Exercise

- Go to the `stacks` directory:

```
cd ~/orchestration-workshop/stacks
```

- Check `registry.yml`:

```
cat registry.yml
```

Deploying our first stack

- All stack manipulation commands start with `docker stack`
- Under the hood, they map to `docker service` commands
- Stacks have a *name* (which also serves as a namespace)
- Stacks are specified with the aforementioned Compose file format version 3

 Exercise

- Deploy our local registry:

```
docker stack deploy registry --compose-file registry.yml
```

Inspecting stacks

- `docker stack ps` shows the detailed state of all services of a stack

Exercise

- Check that our registry is running correctly:

```
docker stack ps registry
```

- Confirm that we get the same output with the following command:

```
docker service ps registry_registry
```

Specifics of stack deployment

Our registry is not *exactly* identical to the one deployed with `docker service create`!

- Each stack gets its own overlay network
- Services of the task are connected to this network
(unless specified differently in the Compose file)
- Services get network aliases matching their name in the Compose file
(just like when Compose brings up an app specified in a v2 file)
- Services are explicitly named `<stack_name>_<service_name>`
- Services and tasks also get an internal label indicating which stack they belong to

Building and pushing stack services

- When using Compose file version 2 and above, you can specify *both* `build` and `image`
- When both keys are present:
 - Compose does "business as usual" (uses `build`)
 - but the resulting image is named as indicated by the `image` key
(instead of `<projectname>_<servicename>:latest`)
 - it can be pushed to a registry with `docker-compose push`
- Example:

```
webfront:
  build: www
  image: myregistry.company.net:5000/webfront
```

Using Compose to build and push images

Exercise

- Try it:

```
docker-compose -f dockercoins.yml build  
docker-compose -f dockercoins.yml push
```

Let's have a look at the `dockercoins.yml` file while this is building and pushing.

```
version: "3"

services:
  rng:
    build: dockercoins/rng
    image: ${REGISTRY-127.0.0.1:5000}/rng:${TAG-latest}
    deploy:
      mode: global
    ...
  redis:
    image: redis
    ...
  worker:
    build: dockercoins/worker
    image: ${REGISTRY-127.0.0.1:5000}/worker:${TAG-latest}
    ...
    deploy:
      replicas: 10
```

Deploying the application

- Now that the images are on the registry, we can deploy our application stack

 Exercise

- Create the application stack:

```
docker stack deploy dockercoins --compose-file dockercoins.yml
```

We can now connect to any of our nodes on port 8000, and we will see the familiar hashing speed graph.

Maintaining multiple environments

There are many ways to handle variations between environments.

- Compose loads `docker-compose.yml` and (if it exists) `docker-compose.override.yml`
- Compose can load alternate file(s) by setting the `-f` flag or the `COMPOSE_FILE` environment variable
- Compose files can *extend* other Compose files, selectively including services:

```
web:  
  extends:  
    file: common-services.yml  
    service: webapp
```

See [this documentation page](#) for more details about these techniques.



Good to know ...

- Compose file version 3 adds the `deploy` section
- Compose file version 3.1 adds support for secrets
- You can re-run `docker stack deploy` to update a stack
- ... But unsupported features will be wiped each time you redeploy (!)

(This will likely be fixed/improved soon)

- `extends` doesn't work with `docker stack deploy`

(But you can use `docker-compose config` to "flatten" your configuration)

Summary

- We've seen how to set up a Swarm
- We've used it to host our own registry
- We've built our app container images
- We've used the registry to host those images
- We've deployed and scaled our application
- We've seen how to use Compose to streamline deployments
- Awesome job, team!

Part 2

Before we start ...

The following exercises assume that you have a 5-nodes Swarm cluster.

If you come here from a previous tutorial and still have your cluster: great!

Otherwise: check [part 1](#) to learn how to setup your own cluster.

We pick up exactly where we left you, so we assume that you have:

- a five nodes Swarm cluster,
- a self-hosted registry,
- DockerCoins up and running.

The next slide has a cheat sheet if you need to set that up in a pinch.

Catching up

Assuming you have 5 nodes provided by [Play-With-Docker](#), do this from `node1`:

```
docker swarm init --advertise-addr eth0
TOKEN=$(docker swarm join-token -q manager)
for N in $(seq 2 5); do
    DOCKER_HOST=tcp://node$N:2375 docker swarm join --token $TOKEN node1:2377
done
git clone git://github.com/jpetazzo/orchestration-workshop
cd orchestration-workshop/stacks
docker stack deploy --compose-file registry.yml registry
docker-compose -f dockercoins.yml build
docker-compose -f dockercoins.yml push
docker stack deploy --compose-file dockercoins.yml dockercoins
```

You should now be able to connect to port 8000 and see the DockerCoins web UI.



Troubleshooting overlay networks

- We want to run tools like `ab` or `httping` on the internal network



Troubleshooting overlay networks

- We want to run tools like `ab` or `httperf` on the internal network
- Ah, if only we had created our overlay network with the `--attachable` flag ...



Troubleshooting overlay networks

- We want to run tools like `ab` or `httping` on the internal network
- Ah, if only we had created our overlay network with the `--attachable` flag ...
- Oh well, let's use this as an excuse to introduce New Ways To Do Things

Breaking into an overlay network

- We will create a dummy placeholder service on our network
- Then we will use `docker exec` to run more processes in this container

Exercise

- Start a "do nothing" container using our favorite Swiss-Army distro:

```
docker service create --network dockercoins_default --name debug \
    --constraint node.hostname==$HOSTNAME alpine sleep 1000000000
```

The `constraint` makes sure that the container will be created on the local node.

Entering the debug container

- Once our container is started (which should be really fast because the alpine image is small), we can enter it (from any node)

Exercise

- Locate the container:

```
docker ps
```

- Enter it:

```
docker exec -ti <containerID> sh
```

Labels

- We can also be fancy and find the ID of the container automatically
- SwarmKit places labels on containers

Exercise

- Get the ID of the container:

```
CID=$(docker ps -q --filter label=com.docker.swarm.service.name=debug)
```

- And enter the container:

```
docker exec -ti $CID sh
```

Installing our debugging tools

- Ideally, you would author your own image, with all your favorite tools, and use it instead of the base `alpine` image
- But we can also dynamically install whatever we need

Exercise

- Install a few tools:

```
apk add --update curl apache2-utils drill
```

Investigating the `rng` service

- First, let's check what `rng` resolves to

 Exercise

- Use drill or nslookup to resolve `rng`:

```
drill rng
```

This give us one IP address. It is not the IP address of a container. It is a virtual IP address (VIP) for the `rng` service.

Investigating the VIP

Exercise

- Try to ping the VIP:

```
ping rng
```

It *should* ping. (But this might change in the future.)

With Engine 1.12: VIPs respond to ping if a backend is available on the same machine.

With Engine 1.13: VIPs respond to ping if a backend is available anywhere.

(Again: this might change in the future.)

What if I don't like VIPs?

- Services can be published using two modes: VIP and DNSRR.
- With VIP, you get a virtual IP for the service, and a load balancer based on IPVS

(By the way, IPVS is totally awesome and if you want to learn more about it in the context of containers, I highly recommend [this talk](#) by [@kobolog](#) at DC15EU!)

- 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
- You change this with `docker service create --endpoint-mode [VIP|DNSRR]`

Looking up VIP backends

- You can also resolve a special name: `tasks.<name>`
- It will give you the IP addresses of the containers for a given service

Exercise

- Obtain the IP addresses of the containers for the `rng` service:

```
drill tasks.rng
```

This should list 5 IP addresses.



Testing and benchmarking our service

- We will check that the service is up with `rng`, then benchmark it with `ab`

Exercise

- Make a test request to the service:

```
curl rng
```

- Open another window, and stop the workers, to test in isolation:

```
docker service update dockercoins_worker --replicas 0
```

Wait until the workers are stopped (check with `docker service ls`) before continuing.



Benchmarking rng

We will send 50 requests, but with various levels of concurrency.

Exercise

- Send 50 requests, with a single sequential client:

```
ab -c 1 -n 50 http://rng/10
```

- Send 50 requests, with fifty parallel clients:

```
ab -c 50 -n 50 http://rng/10
```



Benchmark results for `rng`

- When serving requests sequentially, they each take 100ms
- In the parallel scenario, the latency increased dramatically:
- What about `hasher`?



Benchmarking hasher

We will do the same tests for `hasher`.

The command is slightly more complex, since we need to post random data.

First, we need to put the POST payload in a temporary file.

Exercise

- Install curl in the container, and generate 10 bytes of random data:

```
curl http://rng/10 >/tmp/random
```



Benchmarking hasher

Once again, we will send 50 requests, with different levels of concurrency.

Exercise

- Send 50 requests with a sequential client:

```
ab -c 1 -n 50 -T application/octet-stream -p /tmp/random http://hasher/
```

- Send 50 requests with 50 parallel clients:

```
ab -c 50 -n 50 -T application/octet-stream -p /tmp/random http://hasher/
```



Benchmark results for `hasher`

- The sequential benchmarks takes ~5 seconds to complete
- The parallel benchmark takes less than 1 second to complete
- In both cases, each request takes a bit more than 100ms to complete
- Requests are a bit slower in the parallel benchmark
- It looks like `hasher` is better equipped to deal with concurrency than `rng`



Why?



Why does everything take (at least) 100ms?

`rng` code:

```
21 @app.route("/<int:how_many_bytes>")
22 def rng(how_many_bytes):
23     # Simulate a little bit of delay
24     time.sleep(0.1)
25     return Response()
```

`hasher` code:

```
8 post '/' do
9     # Simulate a bit of delay
10    sleep 0.1
11    content_type 'text/plain'
```



But ...

WHY?!?



Why did we sprinkle this sample app with sleeps?

- Deterministic performance
(regardless of instance speed, CPUs, I/O...)
- Actual code sleeps all the time anyway
- When your code makes a remote API call:
 - it sends a request;
 - it sleeps until it gets the response;
 - it processes the response.



Global scheduling → global debugging

- Traditional approach:
 - log into a node
 - install our Swiss Army Knife (if necessary)
 - troubleshoot things
- Proposed alternative:
 - put our Swiss Army Knife in a container (e.g. [nicolaka/netshoot](#))
 - run tests from multiple locations at the same time

(This becomes very practical with the `docker service log` command, available since 17.05.)



Measuring network conditions on the whole cluster

- Since we have built-in, cluster-wide discovery, it's relatively straightforward to monitor the whole cluster automatically
- [Alexandros Mavrogiannis](#) wrote [Swarm NBT](#), a tool doing exactly that!

Exercise

- Start Swarm NBT:

```
docker run --rm -v inventory:/inventory \
    -v /var/run/docker.sock:/var/run/docker.sock \
    alexmavr/swarm-nbt start
```

Note: in this mode, Swarm NBT connects to the Docker API socket, and issues additional API requests to start all the components it needs.



Viewing network conditions with Prometheus

- Swarm NBT relies on Prometheus to scrape and store data
- We can directly consume the Prometheus endpoint to view telemetry data

Exercise

- Point your browser to any Swarm node, on port 9090
(If you're using Play-With-Docker, click on the (9090) badge)
- In the drop-down, select `icmp_rtt_gauge_seconds`
- Click on "Graph"

You are now seeing ICMP latency across your cluster.

Securing overlay networks

- By default, overlay networks are using plain VXLAN encapsulation
(~Ethernet over UDP, using SwarmKit's control plane for ARP resolution)
- Encryption can be enabled on a per-network basis
(It will use IPSEC encryption provided by the kernel, leveraging hardware acceleration)
- This is only for the `overlay` driver
(Other drivers/plugins will use different mechanisms)

Creating two networks: encrypted and not

- Let's create two networks for testing purposes

Exercise

- Create an "insecure" network:

```
docker network create insecure --driver overlay --attachable
```

- Create a "secure" network:

```
docker network create secure --opt encrypted --driver overlay --attachable
```



⚠ Make sure that you don't typo that option; errors are silently ignored!

Deploying a web server sitting on both networks

- Let's use good old NGINX
- We will attach it to both networks
- We will use a placement constraint to make sure that it is on a different node

Exercise

- Create a web server running somewhere else:

```
docker service create --name web \
    --network secure --network insecure \
    --constraint node.hostname!=node1 \
    nginx
```

Sniff HTTP traffic

- We will use `ngrep`, which allows to grep for network traffic
- We will run it in a container, using host networking to access the host's interfaces

Exercise

- Sniff network traffic and display all packets containing "HTTP":

```
docker run --net host nicolaka/netshoot ngrep -tpd eth0 HTTP
```

Sniff HTTP traffic

- We will use `ngrep`, which allows to grep for network traffic
- We will run it in a container, using host networking to access the host's interfaces

Exercise

- Sniff network traffic and display all packets containing "HTTP":

```
docker run --net host nicolaka/netshoot ngrep -tpd eth0 HTTP
```

Seeing tons of HTTP request? Shutdown your DockerCoins workers:

```
docker service update dockercoins_worker --replicas=0
```

Check that we are, indeed, sniffing traffic

- Let's see if we can intercept our traffic with Google!

Exercise

- Open a new terminal
- Issue an HTTP request to Google (or anything you like):

```
curl google.com
```

The ngrep container will display one # per packet traversing the network interface.

When you do the `curl`, you should see the HTTP request in clear text in the output.



If you are using Play-With-Docker, Vagrant, etc.

- You will probably have *two* network interfaces
- One interface will be used for outbound traffic (to Google)
- The other one will be used for internode traffic
- You might have to adapt/relaunch the `ngrep` command to specify the right one!

Try to sniff traffic across overlay networks

- We will run `curl web` through both secure and insecure networks

Exercise

- Access the web server through the insecure network:

```
docker run --rm --net insecure nicolaka/netshoot curl web
```

- Now do the same through the secure network:

```
docker run --rm --net secure nicolaka/netshoot curl web
```

When you run the first command, you will see HTTP fragments.
However, when you run the second one, only `#` will show up.

Rolling updates

- We want to release a new version of the worker
- We will edit the code ...
- ... build the new image ...
- ... push it to the registry ...
- ... update our service to use the new image



But first...

- Restart the workers

Exercise

- Just scale back to 10 replicas:

```
docker service update dockercoins_worker --replicas 10
```

- Check that they're running:

```
docker service ps dockercoins_worker
```

Making changes

Exercise

- Edit `~/orchestration-workshop/dockercoins/worker/worker.py`
- Locate the line that has a `sleep` instruction
- Increase the `sleep` from `0.1` to `1.0`
- Save your changes and exit

Building and pushing the new image

Exercise

- Go to the `stacks` directory:

```
cd ~/orchestration-workshop/stacks
```

- Build and ship the new image:

```
docker-compose -f dockercoins.yml build  
docker-compose -f dockercoins.yml push
```

Note how the build and push were fast (because caching).

Watching the deployment process

- We will need to open a new terminal for this

Exercise

- Look at our service status:

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

- `docker service ps worker` gives us all tasks
(including the one whose current or desired state is `Shutdown`)
- Then we filter out the tasks whose current **and** desired state is `Shutdown`
- There is also a `--filter` option, but it doesn't allow (yet) to specify that filter

Updating to our new image

- Keep the `watch ...` command running!

 Exercise

- Update our application stack:

```
docker stack deploy dockercoins -c dockercoins.yml
```

If you had stopped the workers earlier, this will automatically restart them.

By default, SwarmKit does a rolling upgrade, one instance at a time.

Changing the upgrade policy

- 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)

Exercise

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

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

The current upgrade will continue at a faster pace.

Changing the policy in the Compose file

- The policy can also be updated in the Compose file
- This is done by adding an `update_config` key under the `deploy` key:

```
deploy:  
  replicas: 10  
  update_config:  
    parallelism: 2  
    delay: 10s
```

Rolling back

- At any time (e.g. before the upgrade is complete), we can rollback:
 - by editing the Compose file and redeploying;
 - or with the special `--rollback` flag



- Try to rollback the service:

```
docker service update dockercoins_worker --rollback
```

What happens with the web UI graph?

The fine print with rollback

- Rollback reverts to the previous service definition
- If we visualize successive updates as a stack:
 - it doesn't "pop" the latest update
 - it "pushes" a copy of the previous update on top
 - ergo, rolling back twice does nothing
- "Service definition" includes rollout cadence
- Each `docker service update` command = a new service definition



Timeline of an upgrade

- SwarmKit will upgrade N instances at a time
(following the `update-parallelism` parameter)
- New tasks are created, and their desired state is set to `Ready`
(this pulls the image if necessary, ensures resource availability, creates the container ... without starting it)
- If the new tasks fail to get to `Ready` state, go back to the previous step
(SwarmKit will try again and again, until the situation is addressed or desired state is updated)
- When the new tasks are `Ready`, it sets the old tasks desired state to `Shutdown`
- When the old tasks are `Shutdown`, it starts the new tasks
- Then it waits for the `update-delay`, and continues with the next batch of instances

Getting task information for a given node

- You can see all the tasks assigned to a node with `docker node ps`
- It shows the *desired state* and *current state* of each task
- `docker node ps` shows info about the current node
- `docker node ps <node_name_or_id>` shows info for another node
- `docker node ps -a` includes stopped and failed tasks

SwarmKit debugging tools

- The SwarmKit repository comes with debugging tools
- They are *low level* tools; not for general use
- We are going to see two of these tools:
 - `swarmctl`, to communicate directly with the SwarmKit API
 - `swarm-rafttool`, to inspect the content of the Raft log

Building the SwarmKit tools

- We are going to install a Go compiler, then download SwarmKit source and build it

Exercise

- Download, compile, and install SwarmKit with this one-liner:

```
docker run -v /usr/local/bin:/go/bin golang \
  go get -v github.com/docker/swarmkit/...
```

Remove `-v` if you don't like verbose things.

Shameless promo: for more Go and Docker love, check [this blog post](#)!

Note: in the unfortunate event of SwarmKit *master* branch being broken, the build might fail. In that case, just skip the Swarm tools section.

Getting cluster-wide task information

- The Docker API doesn't expose this directly (yet)
- But the SwarmKit API does
- We are going to query it with `swarmctl`
- `swarmctl` is an example program showing how to interact with the SwarmKit API

Using `swarmctl`

- The Docker Engine places the SwarmKit control socket in a special path
- You need root privileges to access it

Exercise

- If you are using Play-With-Docker, set the following alias:

```
alias swarmctl='/lib/ld-musl-x86_64.so.1 /usr/local/bin/swarmctl \
--socket /var/run/docker/swarm/control.sock'
```

- Otherwise, set the following alias:

```
alias swarmctl='sudo swarmctl \
--socket /var/run/docker/swarm/control.sock'
```

swarmctl in action

- Let's review a few useful `swarmctl` commands

Exercise

- List cluster nodes (that's equivalent to `docker node ls`):

```
swarmctl node ls
```

- View all tasks across all services:

```
swarmctl task ls
```

swarmctl notes

- SwarmKit is vendored into the Docker Engine
- If you want to use `swarmctl`, you need the exact version of SwarmKit that was used in your Docker Engine
- Otherwise, you might get some errors like:

`Error: grpc: failed to unmarshal the received message proto: wrong wireType = 0`

- With Docker 1.12, the control socket was in `/var/lib/docker/swarm/control.sock`

`swarm-rafttool`

- SwarmKit stores all its important data in a distributed log using the Raft protocol

(This log is also simply called the "Raft log")

- You can decode that log with `swarm-rafttool`
- This is a great tool to understand how SwarmKit works
- It can also be used in forensics or troubleshooting

(But consider it as a *very low level* tool!)

The powers of `swarm-rafttool`

With `swarm-rafttool`, you can:

- view the latest snapshot of the cluster state;
- view the Raft log (i.e. changes to the cluster state);
- view specific objects from the log or snapshot;
- decrypt the Raft data (to analyze it with other tools).

It *cannot* work on live files, so you must stop Docker or make a copy first.

Using `swarm-rafttool`

- First, let's make a copy of the current Swarm data

 Exercise

- If you are using Play-With-Docker, the Docker data directory is `/graph`:

```
cp -r /graph/swarm /swarmdata
```

- Otherwise, it is in the default `/var/lib/docker`:

```
sudo cp -r /var/lib/docker/swarm /swarmdata
```

Dumping the Raft log

- We have to indicate the path holding the Swarm data

(Otherwise `swarm-rafttool` will try to use the live data, and complain that it's locked!)

Exercise

- If you are using Play-With-Docker, you must use the musl linker:

```
/lib/ld-musl-x86_64.so.1 /usr/local/bin/swarm-rafttool -d /swarmdata/ dump-wal
```

- Otherwise, you don't need the musl linker but you need to get root:

```
sudo swarm-rafttool -d /swarmdata/ dump-wal
```

Reminder: this is a very low-level tool, requiring a knowledge of SwarmKit's internals!

Secrets management and encryption at rest

(New in Docker Engine 1.13)

- Secrets management = selectively and securely bring secrets to services
- Encryption at rest = protect against storage theft or prying
- Remember:
 - control plane is authenticated through mutual TLS, certs rotated every 90 days
 - control plane is encrypted with AES-GCM, keys rotated every 12 hours
 - data plane is not encrypted by default (for performance reasons),
but we saw earlier how to enable that with a single flag

Secret management

- Docker has a "secret safe" (secure key→value store)
- You can create as many secrets as you like
- You can associate secrets to services
- Secrets are exposed as plain text files, but kept in memory only (using `tmpfs`)
- Secrets are immutable (at least in Engine 1.13)
- Secrets have a max size of 500 KB

Creating secrets

- Must specify a name for the secret; and the secret itself

Exercise

- Assign **one of the four most commonly used passwords** to a secret called `hackme`:

```
echo love | docker secret create hackme -
```

If the secret is in a file, you can simply pass the path to the file.

(The special path `-` indicates to read from the standard input.)

Creating better secrets

- Picking lousy passwords always leads to security breaches

Exercise

- Let's craft a better password, and assign it to another secret:

```
base64 /dev/urandom | head -c16 | docker secret create arewesecureyet -
```

Note: in the latter case, we don't even know the secret at this point. But Swarm does.

Using secrets

- Secrets must be handed explicitly to services

Exercise

- Create a dummy service with both secrets:

```
docker service create \  
    --secret hackme --secret arewesecureyet \  
    --name dummyservice --mode global \  
    alpine sleep 1000000000
```

We use a global service to make sure that there will be an instance on the local node.

Accessing secrets

- Secrets are materialized on `/run/secrets` (which is an in-memory filesystem)

Exercise

- Find the ID of the container for the dummy service:

```
CID=$(docker ps -q --filter label=com.docker.swarm.service.name=dummyservice)
```

- Enter the container:

```
docker exec -ti $CID sh
```

- Check the files in `/run/secrets`

Rotating secrets

- You can't change a secret

(Sounds annoying at first; but allows clean rollbacks if a secret update goes wrong)

- You can add a secret to a service with `docker service update --secret-add`

(This will redeploy the service; it won't add the secret on the fly)

- You can remove a secret with `docker service update --secret-rm`

- Secrets can be mapped to different names by expressing them with a micro-format:

```
docker service create --secret source=secretname,target=filename
```

Changing our insecure password

- We want to replace our `hackme` secret with a better one

Exercise

- Remove the insecure `hackme` secret:

```
docker service update dummyservice --secret-rm hackme
```

- Add our better secret instead:

```
docker service update dummyservice \
  --secret-add source=arewesecureyet,target=hackme
```

Wait for the service to be fully updated with e.g. `watch docker service ps dummyservice`.

Checking that our password is now stronger

- We will use the power of `docker exec`!

Exercise

- Get the ID of the new container:

```
CID=$(docker ps -q --filter label=com.docker.swarm.service.name=dummyservice)
```

- Check the contents of the secret files:

```
docker exec $CID grep -r . /run/secrets
```

Secrets in practice

- Can be (ab)used to hold whole configuration files if needed
- If you intend to rotate secret `foo`, call it `foo.N` instead, and map it to `foo`
(N can be a serial, a timestamp...)

```
docker service create --secret source=foo.N,target=foo ...
```

- You can update (remove+add) a secret in a single command:

```
docker service update ... --secret-rm foo.M --secret-add source=foo.N,target=foo
```

- For more details and examples, [check the documentation](#)

Improving isolation with User Namespaces

- *Namespaces* are kernel mechanisms to compartmentalize the system
- There are different kind of namespaces: `pid`, `net`, `mnt`, `ipc`, `uts`, and `user`
- For a primer, see "Anatomy of a Container" ([video](#)) ([slides](#))
- The *user namespace* allows to map UIDs between the containers and the host
- As a result, `root` in a container can map to a non-privileged user on the host

Note: even without user namespaces, `root` in a container cannot go wild on the host.
It is mediated by capabilities, cgroups, namespaces, seccomp, LSMs...

User Namespaces in Docker

- Optional feature added in Docker Engine 1.10
- Not enabled by default
- Has to be enabled at Engine startup, and affects all containers
- When enabled, `UID:GID` in containers are mapped to a different range on the host
- Safer than switching to a non-root user (with `-u` or `USER`) in the container
(Since with user namespaces, root escalation maps to a non-privileged user)
- Can be selectively disabled per container by starting them with `-- userns=host`

User Namespaces Caveats

When user namespaces are enabled, containers cannot:

- Use the host's network namespace (with `docker run --network=host`)
- Use the host's PID namespace (with `docker run --pid=host`)
- Run in privileged mode (with `docker run --privileged`)

... Unless user namespaces are disabled for the container, with flag `-- userns=host`

External volume and graph drivers that don't support user mapping might not work.

All containers are currently mapped to the same UID:GID range.

Some of these limitations might be lifted in the future!

Filesystem ownership details

When enabling user namespaces:

- the UID:GID on disk (in the images and containers) has to match the *mapped* UID:GID
- existing images and containers cannot work (their UID:GID would have to be changed)

For practical reasons, when enabling user namespaces, the Docker Engine places containers and images (and everything else) in a different directory.

As a result, if you enable user namespaces on an existing installation:

- all containers and images (and e.g. Swarm data) disappear
- *if a node is a member of a Swarm, it is then kicked out of the Swarm*
- everything will re-appear if you disable user namespaces again

Picking a node

- We will select a node where we will enable user namespaces
- This node will have to be re-added to the Swarm
- All containers and services running on this node will be rescheduled
- Let's make sure that we do not pick the node running the registry!

Exercise

- Check on which node the registry is running:

```
docker service ps registry
```

Pick any other node (noted `nodeX` in the next slides).

Logging into the right Engine

 Exercise

- Log into the right node:

```
ssh nodeX
```

Configuring the Engine

Exercise

- Create a configuration file for the Engine:

```
echo '{"userns-remap": "default"}' | sudo tee /etc/docker/daemon.json
```

- Restart the Engine:

```
kill $(pidof dockerd)
```

Checking that User Namespaces are enabled

Exercise

- Notice the new Docker path:

```
docker info | grep var/lib
```

- Notice the new UID:GID permissions:

```
sudo ls -l /var/lib/docker
```

You should see a line like the following:

```
drwx----- 11 296608 296608 4096 Aug  3 05:11 296608.296608
```

Add the node back to the Swarm

Exercise

- Get our manager token from another node:

```
ssh nodeY docker swarm join-token manager
```

- Copy-paste the join command to the node

Check the new UID:GID

Exercise

- Run a background container on the node:

```
docker run -d --name lockdown alpine sleep 1000000
```

- Look at the processes in this container:

```
docker top lockdown
ps faux
```

Comparing on-disk ownership with/without User Namespaces

Exercise

- Compare the output of the two following commands:

```
docker run alpine ls -l /
docker run -- userns=host alpine ls -l /
```

Comparing on-disk ownership with/without User Namespaces

Exercise

- Compare the output of the two following commands:

```
docker run alpine ls -l /
docker run --userns=host alpine ls -l /
```

In the first case, it looks like things belong to `root:root`.

In the second case, we will see the "real" (on-disk) ownership.

Comparing on-disk ownership with/without User Namespaces

Exercise

- Compare the output of the two following commands:

```
docker run alpine ls -l /
docker run -- userns=host alpine ls -l /
```

In the first case, it looks like things belong to `root:root`.

In the second case, we will see the "real" (on-disk) ownership.

Remember to get back to `node1` when finished!

A reminder about *scope*

- Out of the box, Docker API access is "all or nothing"
- When someone has access to the Docker API, they can access *everything*
- If your developers are using the Docker API to deploy on the dev cluster ...
 - ... and the dev cluster is the same as the prod cluster ...
 - ... it means that your devs have access to your production data, passwords, etc.
- This can easily be avoided

Fine-grained API access control

A few solutions, by increasing order of flexibility:

- Use separate clusters for different security perimeters
(And different credentials for each cluster)

Fine-grained API access control

A few solutions, by increasing order of flexibility:

- Use separate clusters for different security perimeters
(And different credentials for each cluster)
- Add an extra layer of abstraction (sudo scripts, hooks, or full-blown PAAS)

Fine-grained API access control

A few solutions, by increasing order of flexibility:

- Use separate clusters for different security perimeters
(And different credentials for each cluster)
- Add an extra layer of abstraction (sudo scripts, hooks, or full-blown PAAS)
- Enable [authorization plugins](#)
 - each API request is vetted by your plugin(s)
 - by default, the *subject name* in the client TLS certificate is used as user name
 - example: [user and permission management in UCP](#)

Encryption at rest

- Swarm data is always encrypted
- A Swarm cluster can be "locked"
- When a cluster is "locked", the encryption key is protected with a passphrase
- Starting or restarting a locked manager requires the passphrase
- This protects against:
 - theft (stealing a physical machine, a disk, a backup tape...)
 - unauthorized access (to e.g. a remote or virtual volume)
 - some vulnerabilities (like path traversal)

Locking a Swarm cluster

- This is achieved through the `docker swarm update` command

 Exercise

- Lock our cluster:

```
docker swarm update --autolock=true
```

This will display the unlock key. Copy-paste it somewhere safe.

Locked state

- If we restart a manager, it will now be locked

 Exercise

- Restart the local Engine:

```
sudo systemctl restart docker
```

Note: if you are doing the workshop on your own, using nodes that you [provisioned yourself](#) or with [Play-With-Docker](#), you might have to use a different method to restart the Engine.

Checking that our node is locked

- Manager commands (requiring access to encrypted data) will fail
- Other commands are OK

Exercise

- Try a few basic commands:

```
docker ps
docker run alpine echo ♥
docker node ls
```

(The last command should fail, and it will tell you how to unlock this node.)

Checking the state of the node programmatically

- The state of the node shows up in the output of `docker info`

Exercise

- Check the output of `docker info`:

```
docker info
```

- Can't see it? Too verbose? Grep to the rescue!

```
docker info | grep ^Swarm
```

Unlocking a node

- You will need the secret token that we obtained when enabling auto-lock earlier

Exercise

- Unlock the node:

```
docker swarm unlock
```

- Copy-paste the secret token that we got earlier
- Check that manager commands now work correctly:

```
docker node ls
```

Managing the secret key

- If the key is compromised, you can change it and re-encrypt with a new key:

```
docker swarm unlock-key --rotate
```

- If you lost the key, you can get it as long as you have at least one unlocked node:

```
docker swarm unlock-key -q
```

Note: if you rotate the key while some nodes are locked, without saving the previous key, those nodes won't be able to rejoin.

Note: if somebody steals both your disks and your key, ~~you're doomed! Dooooomed!~~ you can block the compromised node with `docker node demote` and `docker node rm`.

Unlocking the cluster permanently

- If you want to remove the secret key, disable auto-lock

Exercise

- Permanently unlock the cluster:

```
docker swarm update --autolock=false
```

Note: if some nodes are in locked state at that moment (or if they are offline/restarting while you disabled autolock), they still need the previous unlock key to get back online.

For more information about locking, you can check the [upcoming documentation](#).

Centralized logging

- We want to send all our container logs to a central place
- If that place could offer a nice web dashboard too, that'd be nice

Centralized logging

- We want to send all our container logs to a central place
- If that place could offer a nice web dashboard too, that'd be nice
- We are going to deploy an ELK stack
- It will accept logs over a GELF socket
- We will update our services to send logs through the GELF logging driver

Setting up ELK to store container logs

*Important foreword: this is not an "official" or "recommended" setup; it is just an example.
We used ELK in this demo because it's a popular setup and we keep being asked about it; but
you will have equal success with Fluent or other logging stacks!*

What we will do:

- Spin up an ELK stack with services
- Gaze at the spiffy Kibana web UI
- Manually send a few log entries using one-shot containers
- Set our containers up to send their logs to Logstash

What's in an ELK stack?

- ELK is three components:
 - ElasticSearch (to store and index log entries)
 - Logstash (to receive log entries from various sources, process them, and forward them to various destinations)
 - Kibana (to view/search log entries with a nice UI)
- The only component that we will configure is Logstash
- We will accept log entries using the GELF protocol
- Log entries will be stored in ElasticSearch,
and displayed on Logstash's stdout for debugging

Setting up ELK

- We need three containers: ElasticSearch, Logstash, Kibana
- We will place them on a common network, `logging`

Exercise

- Create the network:

```
docker network create --driver overlay logging
```

- Create the ElasticSearch service:

```
docker service create --network logging --name elasticsearch elasticsearch:2.4
```

Setting up Kibana

- Kibana exposes the web UI
- Its default port (5601) needs to be published
- It needs a tiny bit of configuration: the address of the ElasticSearch service
- We don't want Kibana logs to show up in Kibana (it would create clutter)
so we tell Logspout to ignore them

Exercise

- Create the Kibana service:

```
docker service create --network logging --name kibana --publish 5601:5601 \
-e ELASTICSEARCH_URL=http://elasticsearch:9200 kibana:4.6
```

Setting up Logstash

- Logstash needs some configuration to listen to GELF messages and send them to ElasticSearch
- We could author a custom image bundling this configuration
- We can also pass the [configuration](#) on the command line

Exercise

- Create the Logstash service:

```
docker service create --network logging --name logstash -p 12201:12201/udp \
    logstash:2.4 -e "$(cat ~/orchestration-workshop/elk/logstash.conf)"
```

Checking Logstash

- Before proceeding, let's make sure that Logstash started properly

 Exercise

- Lookup the node running the Logstash container:

```
docker service ps logstash
```

- Connect to that node

View Logstash logs

Exercise

- Get the ID of the Logstash container:

```
CID=$(docker ps -q --filter label=com.docker.swarm.service.name=logstash)
```

- View the logs:

```
docker logs --follow $CID
```

You should see the heartbeat messages:

```
{      "message" => "ok",
      "host" => "1a4cfb063d13",
      "@version" => "1",
      "@timestamp" => "2016-06-19T00:45:45.273Z"
}
```

Testing the GELF receiver

- In a new window, we will generate a logging message
- We will use a one-off container, and Docker's GELF logging driver

Exercise

- Send a test message:

```
docker run --log-driver=gelf --log-opt=gelf-address=udp://127.0.0.1:12201 \
--rm alpine echo hello
```

The test message should show up in the logstash container logs.

Sending logs from a service

- We were sending from a "classic" container so far; let's send logs from a service instead
- We're lucky: the parameters (`--log-driver` and `--log-opt`) are exactly the same!

Exercise

- Send a test message:

```
docker service create \
    --log-driver gelf --log-opt gelf-address=udp://127.0.0.1:12201 \
    alpine echo hello
```

The test message should show up as well in the logstash container logs.

Sending logs from a service

- We were sending from a "classic" container so far; let's send logs from a service instead
- We're lucky: the parameters (`--log-driver` and `--log-opt`) are exactly the same!

Exercise

- Send a test message:

```
docker service create \
    --log-driver gelf --log-opt gelf-address=udp://127.0.0.1:12201 \
    alpine echo hello
```

The test message should show up as well in the logstash container logs.

In fact, *multiple messages will show up, and continue to show up every few seconds!*

Restart conditions

- By default, if a container exits (or is killed with `docker kill`, or runs out of memory ...), the Swarm will restart it (possibly on a different machine)
- This behavior can be changed by setting the *restart condition* parameter

Exercise

- Change the restart condition so that Swarm doesn't try to restart our container forever:

```
docker service update xxx --restart-condition none
```

Available restart conditions are `none`, `any`, and `on-error`.

You can also set `--restart-delay`, `--restart-max-attempts`, and `--restart-window`.

Connect to Kibana

- The Kibana web UI is exposed on cluster port 5601

 Exercise

- Connect to port 5601 of your cluster
 - if you're using Play-With-Docker, click on the (5601) badge above the terminal
 - otherwise, open [http://\(any-node-address\):5601](http://(any-node-address):5601) with your browser

"Configuring" Kibana

- If you see a status page with a yellow item, wait a minute and reload (Kibana is probably still initializing)
- Kibana should offer you to "Configure an index pattern":
in the "Time-field name" drop down, select "@timestamp", and hit the "Create" button
- Then:
 - click "Discover" (in the top-left corner)
 - click "Last 15 minutes" (in the top-right corner)
 - click "Last 1 hour" (in the list in the middle)
 - click "Auto-refresh" (top-right corner)
 - click "5 seconds" (top-left of the list)
- You should see a series of green bars (with one new green bar every minute)

Updating our services to use GELF

- We will now inform our Swarm to add GELF logging to all our services
- This is done with the `docker service update` command
- The logging flags are the same as before

Exercise

- Enable GELF logging for the `rng` service:

```
docker service update dockercoins_rng \
  --log-driver gelf --log-opt gelf-address=udp://127.0.0.1:12201
```

After ~15 seconds, you should see the log messages in Kibana.

Viewing container logs

- Go back to Kibana
- Container logs should be showing up!
- We can customize the web UI to be more readable

Exercise

- In the left column, move the mouse over the following columns, and click the "Add" button that appears:
 - host
 - container_name
 - message



Don't update stateful services!

- What would have happened if we had updated the Redis service?
- When a service changes, SwarmKit replaces existing container with new ones
- This is fine for stateless services
- But if you update a stateful service, its data will be lost in the process
- If we updated our Redis service, all our DockerCoins would be lost

Important afterword

This is not a "production-grade" setup.

It is just an educational example. We did set up a single ElasticSearch instance and a single Logstash instance.

In a production setup, you need an ElasticSearch cluster (both for capacity and availability reasons). You also need multiple Logstash instances.

And if you want to withstand bursts of logs, you need some kind of message queue: Redis if you're cheap, Kafka if you want to make sure that you don't drop messages on the floor.
Good luck.

If you want to learn more about the GELF driver, have a look at [this blog post](#).

Metrics collection

- We want to gather metrics in a central place
- We will gather node metrics and container metrics
- We want a nice interface to view them (graphs)

Node metrics

- CPU, RAM, disk usage on the whole node
- Total number of processes running, and their states
- Number of open files, sockets, and their states
- I/O activity (disk, network), per operation or volume
- Physical/hardware (when applicable): temperature, fan speed ...
- ... and much more!

Container metrics

- Similar to node metrics, but not totally identical
- RAM breakdown will be different
 - active vs inactive memory
 - some memory is *shared* between containers, and accounted specially
- I/O activity is also harder to track
 - async writes can cause deferred "charges"
 - some page-ins are also shared between containers

For details about container metrics, see:
<http://jpetazzo.github.io/2013/10/08/docker-containers-metrics/>

Tools

We will build *two* different metrics pipelines:

- One based on Intel Snap,
- Another based on Prometheus.

If you're using Play-With-Docker, skip the exercises relevant to Intel Snap (we rely on a SSH server to deploy, and PWD doesn't have that yet).

First metrics pipeline

We will use three open source Go projects for our first metrics pipeline:

- Intel Snap

Collects, processes, and publishes metrics

- InfluxDB

Stores metrics

- Grafana

Displays metrics visually

Snap

- github.com/intelsdi-x/snap
- Can collect, process, and publish metric data
- Doesn't store metrics
- Works as a daemon (snapd) controlled by a CLI (snapctl)
- Offloads collecting, processing, and publishing to plugins
- Does nothing out of the box; configuration required!
- Docs: <https://github.com/intelsdi-x/snap/blob/master/docs/>

InfluxDB

- Snap doesn't store metrics data
- InfluxDB is specifically designed for time-series data
 - CRud vs. CRUD (you rarely if ever update/delete data)
 - orthogonal read and write patterns
 - storage format optimization is key (for disk usage and performance)
- Snap has a plugin allowing to *publish* to InfluxDB

Grafana

- Snap cannot show graphs
- InfluxDB cannot show graphs
- Grafana will take care of that
- Grafana can read data from InfluxDB and display it as graphs

Getting and setting up Snap

- We will install Snap directly on the nodes
- Release tarballs are available from GitHub
- We will use a *global service*
(started on all nodes, including nodes added later)
- This service will download and unpack Snap in /opt and /usr/local
- /opt and /usr/local will be bind-mounted from the host
- This service will effectively install Snap on the hosts

The Snap installer service

- This will get Snap on all nodes

Exercise

```
docker service create --restart-condition=none --mode global \
    --mount type=bind,source=/usr/local/bin,target=/usr/local/bin \
    --mount type=bind,source=/opt,target=/opt centos sh -c '
SNAPVER=v0.16.1-beta
RELEASEURL=https://github.com/intelsdi-x/snap/releases/download/$SNAPVER
curl -sSL $RELEASEURL/snap-$SNAPVER-linux-amd64.tar.gz |
    tar -C /opt -zxf-
curl -sSL $RELEASEURL/snap-plugins-$SNAPVER-linux-amd64.tar.gz |
    tar -C /opt -zxf-
ln -s snap-$SNAPVER /opt/snap
for BIN in snapd snapctl; do ln -s /opt/snap/bin/$BIN /usr/local/bin/$BIN; done
' # If you copy-paste that block, do not forget that final quote ☺
```

First contact with `snapd`

- The core of Snap is `snapd`, the Snap daemon
- Application made up of a REST API, control module, and scheduler module

Exercise

- Start `snapd` with plugin trust disabled and log level set to debug:

```
snapd -t 0 -l 1
```

- More resources:

<https://github.com/intelsdi-x/snap/blob/master/docs/SNAPD.md>

https://github.com/intelsdi-x/snap/blob/master/docs/SNAPD_CONFIGURATION.md

Using `snapctl` to interact with `snappyd`

- Let's load a *collector* and a *publisher* plugins

Exercise

- Open a new terminal
- Load the psutil collector plugin:

```
snapctl plugin load /opt/snap/plugin/snap-plugin-collector-psutil
```

- Load the file publisher plugin:

```
snapctl plugin load /opt/snap/plugin/snap-plugin-publisher-mock-file
```

Checking what we've done

- Good to know: Docker CLI uses `ls`, Snap CLI uses `list`

Exercise

- See your loaded plugins:

```
snapctl plugin list
```

- See the metrics you can collect:

```
snapctl metric list
```

Actually collecting metrics: introducing *tasks*

- To start collecting/processing/publishing metric data, you need to create a *task*
- A *task* indicates:
 - *what* to collect (which metrics)
 - *when* to collect it (e.g. how often)
 - *how* to process it (e.g. use it directly, or compute moving averages)
 - *where* to publish it
- Tasks can be defined with manifests written in JSON or YAML
- Some plugins, such as the Docker collector, allow for wildcards (*) in the metrics "path" (see snap/docker-influxdb.json)
- More resources: <https://github.com/intelsdi-x/snap/blob/master/docs/TASKS.md>

Our first task manifest

```
version: 1
schedule:
  type: "simple" # collect on a set interval
  interval: "1s" # of every 1s
max-failures: 10
workflow:
  collect: # first collect
    metrics: # metrics to collect
      /intel/psutil/load/load1: {}
    config: # there is no configuration
    publish: # after collecting, publish

    plugin_name: "file" # use the file publisher
    config:
      file: "/tmp/snap-psutil-file.log" # write to this file
```

Creating our first task

- The task manifest shown on the previous slide is stored in `snap/psutil-file.yml`.

Exercise

- Create a task using the manifest:

```
cd ~/orchestration-workshop/snap  
snapctl task create -t psutil-file.yml
```

The output should look like the following:

```
Using task manifest to create task  
Task created  
ID: 240435e8-a250-4782-80d0-6fff541facba  
Name: Task-240435e8-a250-4782-80d0-6fff541facba  
State: Running
```

Checking existing tasks

Exercise

- This will confirm that our task is running correctly, and remind us of its task ID

```
sudo snapctl task list
```

The output should look like the following:

ID	NAME	STATE	HIT	MISS	FAIL	CREATED
24043...acba	Task-24043...acba	Running	4	0	0	2:34PM 8-13-2016

Viewing our task dollars at work

- The task is using a very simple publisher, `mock-file`
- That publisher just writes text lines in a file (one line per data point)

Exercise

- Check that the data is flowing indeed:

```
tail -f /tmp/snap-psutil-file.log
```

To exit, hit `^C`

Debugging tasks

- When a task is not directly writing to a local file, use `snapctl task watch`
- `snapctl task watch` will stream the metrics you are collecting to STDOUT

Exercise

```
snapctl task watch <ID>
```

To exit, hit `^C`

Stopping snap

- Our Snap deployment has a few flaws:
 - snapd was started manually
 - it is running on a single node
 - the configuration is purely local

Stopping snap

- Our Snap deployment has a few flaws:
 - snapd was started manually
 - it is running on a single node
 - the configuration is purely local
- We want to change that!

Stopping snap

- Our Snap deployment has a few flaws:
 - snapd was started manually
 - it is running on a single node
 - the configuration is purely local
- We want to change that!
- But first, go back to the terminal where `snapd` is running, and hit `^C`
- All tasks will be stopped; all plugins will be unloaded; Snap will exit

Snap Tribe Mode

- Tribe is Snap's clustering mechanism
- When tribe mode is enabled, nodes can join *agreements*
- When a node in an *agreement* does something (e.g. load a plugin or run a task), other nodes of that agreement do the same thing
- We will use it to load the Docker collector and InfluxDB publisher on all nodes, and run a task to use them
- Without tribe mode, we would have to load plugins and run tasks manually on every node
- More resources: <https://github.com/intelsdi-x/snap/blob/master/docs/TRIBE.md>

Running Snap itself on every node

- Snap runs in the foreground, so you need to use `&` or start it in tmux

 Exercise

- Run the following command *on every node*:

```
snapd -t 0 -l 1 --tribe --tribe-seed node1:6000
```

If you're *not* using Play-With-Docker, there is another way to start Snap!

Starting a daemon through SSH

 Hackety hack ahead!

- We will create a *global service*
- That global service will install a SSH client
- With that SSH client, the service will connect back to its local node
(i.e. "break out" of the container, using the SSH key that we provide)
- Once logged on the node, the service starts snapd with Tribe Mode enabled

Running Snap itself on every node

- I might go to hell for showing you this, but here it goes ...

Exercise

- Start Snapd all over the place:

```
docker service create --name snapd --mode global \
    --mount type=bind,source=$HOME/.ssh/id_rsa,target=/sshkey \
    alpine sh -c "
        apk add --no-cache openssh-client &&
        ssh -o StrictHostKeyChecking=no -i /sshkey docker@172.17.0.1 \
            sudo snapd -t 0 -l 1 --tribe --tribe-seed node1:6000
    " # If you copy-paste that block, don't forget that final quote :-)
```

Remember: this *does not work* with Play-With-Docker (which doesn't have SSH).

Viewing the members of our tribe

- If everything went fine, Snap is now running in tribe mode

 Exercise

- View the members of our tribe:

```
snapctl member list
```

This should show the 5 nodes with their hostnames.

Create an agreement

- We can now create an *agreement* for our plugins and tasks

 Exercise

- Create an agreement; make sure to use the same name all along:

```
snapctl agreement create docker-influxdb
```

The output should look like the following:

Name	Number of Members	plugins	tasks
docker-influxdb	0	0	0

Instruct all nodes to join the agreement

- We dont need another fancy global service!
- We can join nodes from any existing node of the cluster

Exercise

- Add all nodes to the agreement:

```
snapctl member list | tail -n +2 |  
xargs -n1 snapctl agreement join docker-influxdb
```

The last bit of output should look like the following:

Name	Number of Members	plugins	tasks
docker-influxdb	5	0	0

Start a container on every node

- The Docker plugin requires at least one container to be started
- Normally, at this point, you will have at least one container on each node
- But just in case you did things differently, let's create a dummy global service

Exercise

- Create an alpine container on the whole cluster:

```
docker service create --name ping --mode global alpine ping 8.8.8.8
```

Running InfluxDB

- We will create a service for InfluxDB
- We will use the official image
- InfluxDB uses multiple ports:
 - 8086 (HTTP API; we need this)
 - 8083 (admin interface; we need this)
 - 8088 (cluster communication; not needed here)
 - more ports for other protocols (graphite, collectd...)
- We will just publish the first two

Creating the InfluxDB service

Exercise

- Start an InfluxDB service, publishing ports 8083 and 8086:

```
docker service create --name influxdb \
    --publish 8083:8083 \
    --publish 8086:8086 \
    influxdb:0.13
```

Note: this will allow any node to publish metrics data to `localhost:8086`, and it will allow us to access the admin interface by connecting to any node on port 8083.

 Make sure to use InfluxDB 0.13; a few things changed in 1.0 (like, the name of the default retention policy is now "autogen") and this breaks a few things.

Setting up InfluxDB

- We need to create the "snap" database

Exercise

- Open port 8083 with your browser
- Enter the following query in the query box:

```
CREATE DATABASE "snap"
```
- In the top-right corner, select "Database: snap"

Note: the InfluxDB query language *looks like* SQL but it's not.

Load Docker collector and InfluxDB publisher

- We will load plugins on the local node
- Since our local node is a member of the agreement, all other nodes in the agreement will also load these plugins

Exercise

- Load Docker collector:

```
snapctl plugin load /opt/snap/plugin/snap-plugin-collector-docker
```

- Load InfluxDB publisher:

```
snapctl plugin load /opt/snap/plugin/snap-plugin-publisher-influxdb
```

Start a simple collection task

- Again, we will create a task on the local node
- The task will be replicated on other nodes members of the same agreement

Exercise

- Load a task manifest file collecting a couple of metrics on all containers, and sending them to InfluxDB:

```
cd ~/orchestration-workshop/snap  
snapctl task create -t docker-influxdb.json
```

Note: the task description sends metrics to the InfluxDB API endpoint located at 127.0.0.1:8086. Since the InfluxDB container is published on port 8086, 127.0.0.1:8086 always routes traffic to the InfluxDB container.

If things go wrong...

Note: if a task runs into a problem (e.g. it's trying to publish to a metrics database, but the database is unreachable), the task will be stopped.

You will have to restart it manually by running:

```
snapctl task enable <ID>
snapctl task start <ID>
```

This must be done *per node*. Alternatively, you can delete+re-create the task (it will delete+re-create on all nodes).

Check that metric data shows up in InfluxDB

- Let's check existing data with a few manual queries in the InfluxDB admin interface

Exercise

- List "measurements":

```
SHOW MEASUREMENTS
```

(This should show two generic entries corresponding to the two collected metrics.)

- View time series data for one of the metrics:

```
SELECT * FROM "intel/docker/stats/cgroups/cpu_stats/cpu_usage/total_usage"
```

(This should show a list of data points with **time**, **docker_id**, **source**, and **value**.)

Deploy Grafana

- We will use an almost-official image, `grafana/grafana`
- We will publish Grafana's web interface on its default port (3000)

Exercise

- Create the Grafana service:

```
docker service create --name grafana --publish 3000:3000 grafana/grafana:3.1.1
```

Set up Grafana

Exercise

- Open port 3000 with your browser
- Identify with "admin" as the username and password
- Click on the Grafana logo (the orange spiral in the top left corner)
- Click on "Data Sources"
- Click on "Add data source" (green button on the right)

Add InfluxDB as a data source for Grafana

Fill the form exactly as follows:

- Name = "snap"
- Type = "InfluxDB"

In HTTP settings, fill as follows:

- Url = "<http://IP.address.of.any.node:8086>"
- Access = "direct"
- Leave HTTP Auth untouched

In InfluxDB details, fill as follows:

- Database = "snap"
- Leave user and password blank

Finally, click on "add", you should see a green message saying "Success - Data source is working". If you see an orange box (sometimes without a message), it means that you got something wrong. Triple check everything again.

Edit data source

Name	snap	<small> ⓘ</small>	Default	<input type="checkbox"/>
Type	InfluxDB	<small> ⓘ</small>		

Http settings

Url	http://52.28.199.217:8086	<small> ⓘ</small>		
Access	direct	<small> ⓘ</small>		
Http Auth	Basic Auth	<input type="checkbox"/>	With Credentials	<input type="checkbox"/>

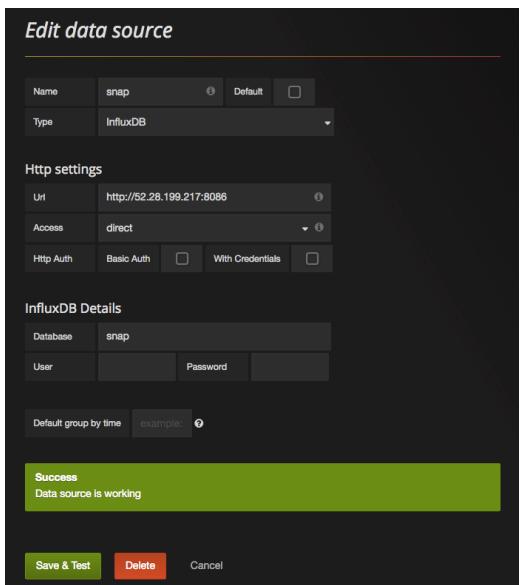
InfluxDB Details

Database	snap		
User	<input type="text"/>	Password	<input type="password"/>

Default group by time example: Ⓢ

Success
Data source is working

Save & Test **Delete** **Cancel**



Create a dashboard in Grafana

Exercise

- Click on the Grafana logo again (the orange spiral in the top left corner)
- Hover over "Dashboards"
- Click "+ New"
- Click on the little green rectangle that appeared in the top left
- Hover over "Add Panel"
- Click on "Graph"

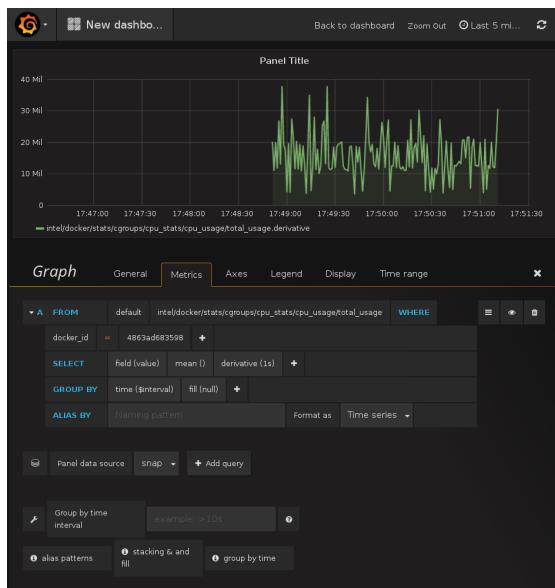
At this point, you should see a sample graph showing up.

Setting up a graph in Grafana

Exercise

- Panel data source: select snap
- Click on the SELECT metrics query to expand it
- Click on "select measurement" and pick CPU usage
- Click on the "+" right next to "WHERE"
- Select "docker_id"
- Select the ID of a container of your choice (e.g. the one running InfluxDB)
- Click on the "+" on the right of the "SELECT" line
- Add "derivative"
- In the "derivative" option, select "1s"
- In the top right corner, click on the clock, and pick "last 5 minutes"

Congratulations, you are viewing the CPU usage of a single container!



Before moving on ...

- Leave that tab open!
- We are going to setup *another* metrics system
- ... And then compare both graphs side by side

Prometheus vs. Snap

- Prometheus is another metrics collection system
- Snap *pushes* metrics; Prometheus *pulls* them

Prometheus components

- The *Prometheus server* pulls, stores, and displays metrics
- Its configuration defines a list of *exporter* endpoints
(that list can be dynamic, using e.g. Consul, DNS, Etcd...)
- The exporters expose metrics over HTTP using a simple line-oriented format
(An optimized format using protobuf is also possible)

It's all about the /metrics

- This is what the *node exporter* looks like:

<http://demo.robustperception.io:9100/metrics>

- Prometheus itself exposes its own internal metrics, too:

<http://demo.robustperception.io:9090/metrics>

- A *Prometheus server* will *scrape* URLs like these

(It can also use protobuf to avoid the overhead of parsing line-oriented formats!)

Collecting metrics with Prometheus on Swarm

- We will run two *global services* (i.e. scheduled on all our nodes):
 - the Prometheus *node exporter* to get node metrics
 - Google's cAdvisor to get container metrics
- We will run a Prometheus server to scrape these exporters
- The Prometheus server will be configured to use DNS service discovery
- We will use `tasks.<servicename>` for service discovery
- All these services will be placed on a private internal network

Creating an overlay network for Prometheus

- This is the easiest step ☺



- Exercise
- Create an overlay network:

```
docker network create --driver overlay prom
```

Running the node exporter

- The node exporter *should* run directly on the hosts
- However, it can run from a container, if configured properly
(it needs to access the host's filesystems, in particular /proc and /sys)

Exercise

- Start the node exporter:

```
docker service create --name node --mode global --network prom \
  --mount type=bind,source=/proc,target=/host/proc \
  --mount type=bind,source=/sys,target=/host/sys \
  --mount type=bind,source=/,target=/rootfs \
  prom/node-exporter \
    -collector.procfs /host/proc \
    -collector.sysfs /host/proc \
    -collector.filesystem.ignored-mount-points "^(sys|proc|dev|host|etc)(\$|/)"
```

Running cAdvisor

- Likewise, cAdvisor *should* run directly on the hosts
- But it can run in containers, if configured properly

Exercise

- Start the cAdvisor collector:

```
docker service create --name cadvisor --network prom --mode global \
  --mount type=bind,source=/,target=/rootfs \
  --mount type=bind,source=/var/run,target=/var/run \
  --mount type=bind,source=/sys,target=/sys \
  --mount type=bind,source=/var/lib/docker,target=/var/lib/docker \
  google/cadvisor:latest
```

Configuring the Prometheus server

This will be our configuration file for Prometheus:

```
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'node'
    dns_sd_configs:
      - names: ['tasks.node']
        type: 'A'
        port: 9100
  - job_name: 'cadvisor'
    dns_sd_configs:
      - names: ['tasks.cadvisor']
        type: 'A'
        port: 8080
```

Passing the configuration to the Prometheus server

- We need to provide our custom configuration to the Prometheus server
- The easiest solution is to create a custom image bundling this configuration
- We will use a very simple Dockerfile:

```
FROM prom/prometheus:v1.4.1
COPY prometheus.yml /etc/prometheus/prometheus.yml
```

(The configuration file, and the Dockerfile, are in the `prom` subdirectory)

- We will build this image, and push it to our local registry
- Then we will create a service using this image

Building our custom Prometheus image

- We will use the local registry started previously on 127.0.0.1:5000

Exercise

- Build the image using the provided Dockerfile:

```
docker build -t 127.0.0.1:5000/prometheus ~/orchestration-workshop/prom
```

- Push the image to our local registry:

```
docker push 127.0.0.1:5000/prometheus
```

Running our custom Prometheus image

- That's the only service that needs to be published

(If we want to access Prometheus from outside!)

Exercise

- Start the Prometheus server:

```
docker service create --network prom --name prom \
    --publish 9090:9090 127.0.0.1:5000/prometheus
```

Checking our Prometheus server

- First, let's make sure that Prometheus is correctly scraping all metrics

 Exercise

- Open port 9090 with your browser
- Click on "status", then "targets"

You should see 11 endpoints (5 cAdvisor, 5 node, 1 prometheus).

Their state should be "UP".

Displaying metrics directly from Prometheus

- This is easy ... if you are familiar with PromQL

Exercise

- Click on "Graph", and in "expression", paste the following:

```
sum by (container_label_com_docker_swarm_node_id) (
  irate(
    container_cpu_usage_seconds_total{
      container_label_com_docker_swarm_service_name="dockercoins_worker"
    }[1m]
  )
)
```

- Click on the blue "Execute" button and on the "Graph" tab just below

Building the query from scratch

- We are going to build the same query from scratch
- This doesn't intend to be a detailed PromQL course
- This is merely so that you (I) can pretend to know how the previous query works
so that your coworkers (you) can be suitably impressed (or not)

(Or, so that we can build other queries if necessary, or adapt if cAdvisor, Prometheus, or anything else changes and requires editing the query!)

Displaying a raw metric for *all* containers

- Click on the "Graph" tab on top

This takes us to a blank dashboard

- Click on the "Insert metric at cursor" drop down, and select
`container_cpu_usage_seconds_total`

This puts the metric name in the query box

- Click on "Execute"

This fills a table of measurements below

- Click on "Graph" (next to "Console")

This replaces the table of measurements with a series of graphs (after a few seconds)

Selecting metrics for a specific service

- Hover over the lines in the graph

(Look for the ones that have labels like `container_label_com_docker_...`)

- Edit the query, adding a condition between curly braces:

```
container_cpu_usage_seconds_total{container_label_com_docker_swarm_service_name="dockercoints_worker"}
```

- Click on "Execute"

Now we should see one line per CPU per container

- If you want to select by container ID, you can use a regex match: `id=~"/docker/c4bf.*"`
- You can also specify multiple conditions by separating them with commas

Turn counters into rates

- What we see is the total amount of CPU used (in seconds)
- We want to see a *rate* (CPU time used / real time)
- To get a moving average over 1 minute periods, enclose the current expression within:

```
rate ( ... { ... } [1m] )
```

This should turn our steadily-increasing CPU counter into a wavy graph

- To get an instantaneous rate, use `irate` instead of `rate`

(The time window is then used to limit how far behind to look for data if data points are missing in case of scrape failure; see [here](#) for more details!)

This should show spikes that were previously invisible because they were smoothed out

Aggregate multiple data series

- We have one graph per CPU per container; we want to sum them
- Enclose the whole expression within:

`sum (...)`

We now see a single graph

Collapse dimensions

- If we have multiple containers we can also collapse just the CPU dimension:

```
sum without (cpu) ( ... )
```

This shows the same graph, but preserves the other labels

- Congratulations, you wrote your first PromQL expression from scratch!

(I'd like to thank [Johannes Ziemke](#) and [Julius Volz](#) for their help with Prometheus!)

Comparing Snap and Prometheus data

- If you haven't setup Snap, InfluxDB, and Grafana, skip this section
- If you have closed the Grafana tab, you might have to re-setup a new dashboard
(Unless you saved it before navigating it away)
- To re-do the setup, just follow again the instructions from the previous chapter

Add Prometheus as a data source in Grafana

Exercise

- In a new tab, connect to Grafana (port 3000)
- Click on the Grafana logo (the orange spiral in the top-left corner)
- Click on "Data Sources"
- Click on the green "Add data source" button

We see the same input form that we filled earlier to connect to InfluxDB.

Connecting to Prometheus from Grafana

Exercise

- Enter "prom" in the name field
- Select "Prometheus" as the source type
- Enter [http://\(IP.address.of.any.node\):9090](http://(IP.address.of.any.node):9090) in the Url field
- Select "direct" as the access method
- Click on "Save and test"

Again, we should see a green box telling us "Data source is working."

Otherwise, double-check every field and try again!

Adding the Prometheus data to our dashboard

Exercise

- Go back to the the tab where we had our first Grafana dashboard
- Click on the blue "Add row" button in the lower right corner
- Click on the green tab on the left; select "Add panel" and "Graph"

This takes us to the graph editor that we used earlier.

Querying Prometheus data from Grafana

The editor is a bit less friendly than the one we used for InfluxDB.

Exercise

- Select "prom" as Panel data source
- Paste the query in the query field:

```
sum without (cpu, id) ( rate (
  container_cpu_usage_seconds_total{
    container_label_com_docker_swarm_service_name="influxdb"}[1m] ) )
```

- Click outside of the query field to confirm
- Close the row editor by clicking the "X" in the top right area

Interpreting results

- The two graphs *should* be similar
- Protip: align the time references!

Exercise

- Click on the clock in the top right corner
- Select "last 30 minutes"
- Click on "Zoom out"
- Now press the right arrow key (hold it down and watch the CPU usage increase!)

Adjusting units is left as an exercise for the reader.

More resources on container metrics

- [Prometheus, a Whirlwind Tour](#), an original overview of Prometheus
- [Docker Swarm & Container Overview](#), a custom dashboard for Grafana
- [Gathering Container Metrics](#), a blog post about cgroups
- [The Prometheus Time Series Database](#), a talk explaining why custom data storage is necessary for metrics

Dealing with stateful services

- First of all, you need to make sure that the data files are on a *volume*
- Volumes are host directories that are mounted to the container's filesystem
- These host directories can be backed by the ordinary, plain host filesystem ...
- ... Or by distributed/networked filesystems
- In the latter scenario, in case of node failure, the data is safe elsewhere ...
- ... And the container can be restarted on another node without data loss

Building a stateful service experiment

- We will use Redis for this example
- We will expose it on port 10000 to access it easily

Exercise

- Start the Redis service:

```
docker service create --name stateful -p 10000:6379 redis
```

- Check that we can connect to it:

```
docker run --net host --rm redis redis-cli -p 10000 info server
```

Accessing our Redis service easily

- Typing that whole command is going to be tedious

Exercise

- Define a shell alias to make our lives easier:

```
alias redis='docker run --net host --rm redis redis-cli -p 10000'
```

- Try it:

```
redis info server
```

Basic Redis commands

Exercise

- Check that the `foo` key doesn't exist:

```
redis get foo
```

- Set it to `bar`:

```
redis set foo bar
```

- Check that it exists now:

```
redis get foo
```

Local volumes vs. global volumes

- Global volumes exist in a single namespace
- A global volume can be mounted on any node
(bar some restrictions specific to the volume driver in use; e.g. using an EBS-backed volume on a GCE/EC2 mixed cluster)
- Attaching a global volume to a container allows to start the container anywhere
(and retain its data wherever you start it!)
- Global volumes require extra *plugins* (Flocker, Portworx...)
- Docker doesn't come with a default global volume driver at this point
- Therefore, we will fall back on *local volumes*

Local volumes

- We will use the default volume driver, `local`
- As the name implies, the `local` volume driver manages *local* volumes
- Since local volumes are (duh!) *local*, we need to pin our container to a specific host
- We will do that with a *constraint*

 Exercise

- Add a placement constraint to our service:

```
docker service update stateful --constraint-add node.hostname==$HOSTNAME
```

Where is our data?

- If we look for our `foo` key, it's gone!

Exercise

- Check the `foo` key:

```
redis get foo
```

- Adding a constraint caused the service to be redeployed:

```
docker service ps stateful
```

Note: even if the constraint ends up being a no-op (i.e. not moving the service), the service gets redeployed. This ensures consistent behavior.

Setting the key again

- Since our database was wiped out, let's populate it again

Exercise

- Set foo again:

```
redis set foo bar
```

- Check that it's there:

```
redis get foo
```

Service updates cause containers to be replaced

- Let's try to make a trivial update to the service and see what happens

Exercise

- Set a memory limit to our Redis service:

```
docker service update stateful --limit-memory 100M
```

- Try to get the `foo` key one more time:

```
redis get foo
```

The key is blank again!

Service volumes are ephemeral by default

- Let's highlight what's going on with volumes!

Exercise

- Check the current list of volumes:

```
docker volume ls
```

- Carry a minor update to our Redis service:

```
docker service update stateful --limit-memory 200M
```

Again: all changes trigger the creation of a new task, and therefore a replacement of the existing container; even when it is not strictly technically necessary.

The data is gone again

- What happened to our data?

 Exercise

- The list of volumes is slightly different:

```
docker volume ls
```

(You should see one extra volume.)

Assigning a persistent volume to the container

- Let's add an explicit volume mount to our service, referencing a named volume

Exercise

- Update the service with a volume mount:

```
docker service update stateful \
  --mount-add type=volume,source=foobarstore,target=/data
```

- Check the new volume list:

```
docker volume ls
```

Note: the `local` volume driver automatically creates volumes.

Checking that persistence actually works across service updates

Exercise

- Store something in the `foo` key:

```
redis set foo barbar
```

- Update the service with yet another trivial change:

```
docker service update stateful --limit-memory 300M
```

- Check that `foo` is still set:

```
redis get foo
```

Recap

- The service must commit its state to disk when being shutdown*
- (Shutdown = being sent a `TERM` signal)
- The state must be written on files located on a volume
- That volume must be specified to be persistent
- If using a local volume, the service must also be pinned to a specific node

(And losing that node means losing the data, unless there are other backups)

*If you customize Redis configuration, make sure you persist data correctly!
It's easy to make that mistake — **Trust me!**

Cleaning up

Exercise

- Remove the stateful service:

```
docker service rm stateful
```

- Remove the associated volume:

```
docker volume rm foobarstore
```

Note: we could keep the volume around if we wanted.

Should I run stateful services in containers?

Should I run stateful services in containers?

Depending whom you ask, they'll tell you:

Should I run stateful services in containers?

Depending whom you ask, they'll tell you:

- certainly not, heathen!

Should I run stateful services in containers?

Depending whom you ask, they'll tell you:

- certainly not, heathen!
- we've been running a few thousands PostgreSQL instances in containers ...
for a few years now ... in production ... is that bad?

Should I run stateful services in containers?

Depending whom you ask, they'll tell you:

- certainly not, heathen!
- we've been running a few thousands PostgreSQL instances in containers ...
for a few years now ... in production ... is that bad?
- what's a container?

Should I run stateful services in containers?

Depending whom you ask, they'll tell you:

- certainly not, heathen!
- we've been running a few thousands PostgreSQL instances in containers ...
for a few years now ... in production ... is that bad?
- what's a container?

Perhaps a better question would be:

"Should I run stateful services?"

Should I run stateful services in containers?

Depending whom you ask, they'll tell you:

- certainly not, heathen!
- we've been running a few thousands PostgreSQL instances in containers ...
for a few years now ... in production ... is that bad?
- what's a container?

Perhaps a better question would be:

"Should I run stateful services?"

- is it critical for my business?
- is it my value-add?
- or should I find somebody else to run them for me?



Controlling Docker from a container

- In a local environment, just bind-mount the Docker control socket:

```
docker run -ti -v /var/run/docker.sock:/var/run/docker.sock docker
```

- Otherwise, you have to:
 - set `DOCKER_HOST`,
 - set `DOCKER_TLS_VERIFY` and `DOCKER_CERT_PATH` (if you use TLS),
 - copy certificates to the container that will need API access.

More resources on this topic:

- [Do not use Docker-in-Docker for CI](#)
- [One container to rule them all](#)



Bind-mounting the Docker control socket

- In Swarm mode, bind-mounting the control socket gives you access to the whole cluster
- You can tell Docker to place a given service on a manager node, using constraints:

```
docker service create \  
  --mount source=/var/run/docker.sock,type=bind,target=/var/run/docker.sock \  
  --name autoscaler --constraint node.role==manager ...
```



Constraints and global services

(New in Docker Engine 1.13)

- By default, global services run on *all* nodes

```
docker service create --mode global ...
```

- You can specify constraints for global services
- These services will run only on the node satisfying the constraints
- For instance, this service will run on all manager nodes:

```
docker service create --mode global --constraint node.role==manager ...
```



Constraints and dynamic scheduling

(New in Docker Engine 1.13)

- If constraints change, services are started/stopped accordingly
(e.g., `--constraint node.role==manager` and nodes are promoted/demoted)
- This is particularly useful with labels:

```
docker node update node1 --label-add defcon=five
docker service create --constraint node.labels.defcon==five ...
docker node update node2 --label-add defcon=five
docker node update node1 --label-rm defcon=five
```



Shortcomings of dynamic scheduling

⚠ If a service becomes "unschedulable" (constraints can't be satisfied):

- It won't be scheduled automatically when constraints are satisfiable again
- You will have to update the service; you can do a no-op update with:

```
docker service update ... --force
```

⚠ Docker will silently ignore attempts to remove a non-existent label or constraint

- It won't warn you if you typo when removing a label or constraint!



Node management

- SwarmKit allows to change (almost?) everything on-the-fly
- Nothing should require a global restart



Node availability

```
docker node update <node-name> --availability <active|pause|drain>
```

- Active = schedule tasks on this node (default)
- Pause = don't schedule new tasks on this node; existing tasks are not affected

You can use it to troubleshoot a node without disrupting existing tasks

It can also be used (in conjunction with labels) to reserve resources

- Drain = don't schedule new tasks on this node; existing tasks are moved away

This is just like crashing the node, but containers get a chance to shutdown cleanly



Managers and workers

- Nodes can be promoted to manager with `docker node promote`
- Nodes can be demoted to worker with `docker node demote`
- This can also be done with `docker node update <node> --role <manager|worker>`
- Reminder: this has to be done from a manager node
(workers cannot promote themselves)



Removing nodes

- You can leave Swarm mode with `docker swarm leave`
- Nodes are drained before being removed (i.e. all tasks are rescheduled somewhere else)
- Managers cannot leave (they have to be demoted first)
- After leaving, a node still shows up in `docker node ls` (in `Down` state)
- When a node is `Down`, you can remove it with `docker node rm` (from a manager node)



Join tokens and automation

- If you have used Docker 1.12-RC: join tokens are now mandatory!
- You cannot specify your own token (SwarmKit generates it)
- If you need to change the token: `docker swarm join-token --rotate ...`
- To automate cluster deployment:
 - have a seed node do `docker swarm init` if it's not already in Swarm mode
 - propagate the token to the other nodes (secure bucket, factor, ohai...)



Disk space management: docker system df

- Shows disk usage for images, containers, and volumes
- Breaks down between *active* and *reclaimable* categories

Exercise

- Check how much disk space is used at the end of the workshop:

```
docker system df
```

Note: `docker system` is new in Docker Engine 1.13.



Reclaiming unused resources: `docker system prune`

- Removes stopped containers
- Removes dangling images (that don't have a tag associated anymore)
- Removes orphaned volumes
- Removes empty networks

Exercise

- Try it:

```
docker system prune -f
```

Note: `docker system prune -a` will also remove *unused* images.



What's next?

(What to expect in future versions of this workshop)



Implemented and stable, but out of scope

- [Docker Content Trust](#) and [Notary](#) (image signature and verification)
- Image security scanning (many products available, Docker Inc. and 3rd party)
- [Docker Cloud](#) and [Docker Datacenter](#) (commercial offering with node management, secure registry, CI/CD pipelines, all the bells and whistles)
- Network and storage plugins



Work in progress

- Stabilize Compose/Swarm integration
- Refine Snap deployment
- Healthchecks
- Demo at least one volume plugin
(bonus points if it's a distributed storage system)
- (your favorite feature here)

Reminder: there is a tag for each iteration of the content in the Github repository.

It makes it easy to come back later and check what has changed since you did it!

Thank you!