# Kubernetes for Beginners

## Start the cluster

First step is to initialize the cluster in the first terminal:

```
kubeadm init --apiserver-advertise-address $(hostname -i)
```

That will take a couple of minutes, during which time you'll see a lot of activity in the terminal.

You will see something like this at the end:

```
Your Kubernetes master has initialized successfully!


To start using your cluster, you need to run (as a regular user):


  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config


You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  http://kubernetes.io/docs/admin/addons/


You can now join any number of machines by running the following on each node
as root:


kubeadm join --token SOMETOKEN SOMEIPADDRESS --discovery-token-ca-cert-hash SOME
SHAHASH
```

Copy the whole line that starts `kubeadm join` from the first terminal and paste it into the second terminal. You should see something like this:

```
kubeadm join --token a146c9.9421a0d62a0611f4 172.26.0.2:6443 --discovery-token-ca-
cert-hash sha256:9a4dc07bd8ac596336ecee6ce0928b3500174037c07a38a03bebef25e97c4db5
Initializing machine ID from random generator.
[kubeadm] WARNING: kubeadm is in beta, please do not use it for production cluster
s.
[preflight] Skipping pre-flight checks
[discovery] Trying to connect to API Server "172.26.0.2:6443"
```

```
[discovery] Created cluster-info discovery client, requesting info from "https://1
72.26.0.2:6443"

[discovery] Requesting info from "https://172.26.0.2:6443" again to validate TLS a
gainst the pinned public key

[discovery] Cluster info signature and contents are valid and TLS certificate vali
dates against pinned roots, will use API Server "172.26.0.2:6443"

[discovery] Successfully established connection with API Server "172.26.0.2:6443"

[bootstrap] Detected server version: v1.8.11

[bootstrap] The server supports the Certificates API (certificates.k8s.io/v1beta1)


Node join complete:

* Certificate signing request sent to master and response

  received.

* Kubelet informed of new secure connection details.


Run 'kubectl get nodes' on the master to see this machine join.
```

That means you're almost ready to go. Last you just have to initialize your cluster networking in the first terminal:

```
   kubectl apply -n kube-system -f \

    "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 |tr
-d '\n')"
```

You'll see an output like this:

```
  kubectl apply -n kube-system -f \

  >     "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64
|tr -d '\n')"

  serviceaccount "weave-net" created

  clusterrole "weave-net" created

  clusterrolebinding "weave-net" created

  role "weave-net" created

  rolebinding "weave-net" created

  daemonset "weave-net" created
```

Your cluster is set up!

# 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 generate a few random bytes
  - `worker` feeds these bytes into `hasher`
  - and repeat forever!
  - every second, `worker` updates `redis` to indicate how many loops were done
  - `webui` queries `redis`, and computes and exposes “hashing speed” in your browser

# Getting the application source code

We’ve created a sample application to run for parts of the workshop. The application is in the dockercoins repository.

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

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

```
git clone https://github.com/dockersamples/dockercoins
```

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

# Running the application

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

```
cd ~/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

## Connecting to the web UI

- The `webui` container exposes a web dashboard; let's view it
- With a web browser, connect to `node1` here on port 8000 (created when you ran the application)

## Clean up

Before moving on, let's turn everything off by typing `Ctrl-C`.

# Kubernetes concepts

- Kubernetes is a container management system
- It runs and manages containerized applications on a cluster
- What does that really mean?

## Basic things we can ask Kubernetes to do

- Start 5 containers using image `atseashop/api:v1.3`
- Place an internal load balancer in front of these containers
- Start 10 containers using image `atseashop/webfront:v1.3`
- Place a public load balancer in front of these containers
- It's Black Friday (or Christmas), traffic spikes, grow our cluster and add containers
- New release! Replace my containers with the new image `atseashop/webfront:v1.4`
- Keep processing requests during the upgrade; update my containers one at a time

## Other things that Kubernetes can do for us

- Basic autoscaling
- Blue/green deployment, canary deployment
- Long running services, but also batch (one-off) jobs
- Overcommit our cluster and evict low-priority jobs
- Run services with stateful data (databases etc.)
- Fine-grained access control defining what can be done by whom on which resources
- Integrating third party services (service catalog)
- Automating complex tasks (operators)

# Kubernetes architecture

## Kubernetes architecture: the master

- The Kubernetes logic (its "brains") is a collection of services:

    - the API server (our point of entry to everything!)
    - core services like the scheduler and controller manager
    - etcd (a highly available key/value store; the “database” of Kubernetes)

*Together, these services form what is called the “master”

- These services can run straight on a host, or in containers (that’s an implementation detail)
- etcd can be run on separate machines (first schema) or co-located (second schema)
- We need at least one master, but we can have more (for high availability)

## Kubernetes architecture: the nodes

- The nodes executing our containers run another collection of services:
    - a container Engine (typically Docker)
    - kubelet (the “node agent”)
    - kube-proxy (a necessary but not sufficient network component)
- Nodes were formerly called “minions”
- It is customary to not run apps on the node(s) running master components (Except when using small development clusters)

# Kubernetes resources

- The Kubernetes API defines a lot of objects called resources
- These resources are organized by type, or Kind (in the API)
- A few common resource types are:
    - node (a machine — physical or virtual — in our cluster)
    - pod (group of containers running together on a node)
    - service (stable network endpoint to connect to one or multiple containers)
    - namespace (more-or-less isolated group of things)
    - secret (bundle of sensitive data to be passed to a container)
- And much more! (We can see the full list by running kubectl get)

# Declarative vs imperative

- Our container orchestrator puts a very strong emphasis on being declarative
- Declarative:

  *I would like a cup of tea.*
- Imperative:

  *Boil some water. Pour it in a teapot. Add tea leaves. Steep for a while. Serve in cup.*
- Declarative seems simpler at first …
- … As long as you know how to brew tea

### What declarative would really be:

*I want a cup of tea, obtained by pouring an infusion of tea leaves in a cup.*

*An infusion is obtained by letting the object steep a few minutes in hot water.*

*Hot liquid is obtained by pouring it in an appropriate container and setting it on a stove.*

*Ah, finally, containers! Something we know about. Let's get to work, shall we?*

## Summary of declarative vs imperative

- Imperative systems:
  - simpler
  - if a task is interrupted, we have to restart from scratch
- Declarative systems:
  - if a task is interrupted (or if we show up to the party half-way through), we can figure out what's missing and do only what's necessary
  - we need to be able to *observe* the system
  - … and compute a "diff" between *what we have and what we want*

## Declarative vs imperative in Kubernetes

- Virtually everything we create in Kubernetes is created from a `spec`
- Watch for the `spec` fields in the YAML files later!
- The `spec` describes *how we want the thing to be*
- Kubernetes will *reconcile* the current state with the spec (technically, this is done by a number of *controllers*)
- When we want to change some resource, we update the `spec`
- Kubernetes will then *converge* that resource

# Kubernetes network model

- TL,DR:

  *Our cluster (nodes and pods) is one big flat IP network.*
- In detail:
  - all nodes must be able to reach each other, without NAT
  - all pods must be able to reach each other, without NAT
  - pods and nodes must be able to reach each other, without NAT
  - each pod is aware of its IP address (no NAT)
- Kubernetes doesn't mandate any particular implementation

# Kubernetes network model: the good

- Everything can reach everything
- No address translation

- No port translation
- No new protocol
- Pods cannot move from a node to another and keep their IP address
- IP addresses don't have to be "portable" from a node to another (We can use e.g. a subnet per node and use a simple routed topology)
- The specification is simple enough to allow many various implementations

# Kubernetes network model: the less good

- Everything can reach everything
  - if you want security, you need to add network policies
  - the network implementation that you use needs to support them
- There are literally dozens of implementations out there (15 are listed in the Kubernetes documentation)
- It looks like you have a level 3 network, but it's only level 4 (The spec requires UDP and TCP, but not port ranges or arbitrary IP packets)
- `kube-proxy` is on the data path when connecting to a pod or container, and it's not particularly fast (relies on userland proxying or iptables)

# Kubernetes network model: in practice

- The nodes that we are using have been set up to use Weave
- We don't endorse Weave in a particular way, it just Works For Us
- Don't worry about the warning about kube-proxy performance
- Unless you:
  - routinely saturate 10G network interfaces
  - count packet rates in millions per second
  - run high-traffic VOIP or gaming platforms
  - do weird things that involve millions of simultaneous connections (in which case you're already familiar with kernel tuning)

# First contact with `kubectl`

- `kubectl` is (almost) the only tool we'll need to talk to Kubernetes
- It is a rich CLI tool around the Kubernetes API (Everything you can do with `kubectl`, you can do directly with the API)
- You can also use the `--kubeconfig` flag to pass a config file
- Or directly `--server`, `--user`, etc.
- `kubectl` can be pronounced "Cube C T L", "Cube cuttle", "Cube cuddle"…

## `kubectl get`

- Let's look at our Node resources with kubectl get!
- Look at the composition of our cluster:

```
kubectl get node
```

- These commands are equivalent

```
kubectl get no
kubectl get node
kubectl get nodes
```

## Obtaining machine-readable output

- `kubectl get` can output JSON, YAML, or be directly formatted
- Give us more info about the nodes:

```
kubectl get nodes -o wide
```

- Let's have some YAML:

```
kubectl get no -o yaml
```

See that kind: List at the end? It's the type of our result!

## (Ab)using `kubectl` and `jq`

- It's super easy to build custom reports
- Show the capacity of all our nodes as a stream of JSON objects:

```
kubectl get nodes -o json |
     jq ".items[] | {name:.metadata.name} + .status.capacity"
```

## What's available?

- `kubectl` has pretty good introspection facilities
- We can list all available resource types by running `kubectl get`
- We can view details about a resource with:

```
kubectl describe type/name
kubectl describe type name
```

- We can view the definition for a resource type with:

```
kubectl explain type
```

Each time, `type` can be singular, plural, or abbreviated type name.

# Services

- A service is a stable endpoint to connect to “something” (In the initial proposal, they were called “portals”)
- List the services on our cluster:

```
kubectl get services
```

This would also work:

```
kubectl get svc
```

There is already one service on our cluster: the Kubernetes API itself.

# ClusterIP services

- A `ClusterIP` service is internal, available from the cluster only
- This is useful for introspection from within containers
- Try to connect to the API.
  - `-k` is used to skip certificate verification
  - Make sure to replace 10.96.0.1 with the CLUSTER-IP shown by `$ kubectl get svc`

```
curl -k https://10.96.0.1
```

The error that we see is expected: the Kubernetes API requires authentication.

# Listing running containers

- Containers are manipulated through pods
- A pod is a group of containers:
  - running together (on the same node)
  - sharing resources (RAM, CPU; but also network, volumes)
- List pods on our cluster:

```
kubectl get pods
```

*These are not the pods you’re looking for*. But where are they?!?

# Namespaces

- Namespaces allow us to segregate resources
- List the namespaces on our cluster with one of these commands:

```
kubectl get namespaces
```

either of these would work as well:

```
kubectl get namespace
kubectl get ns
```

*You know what … This `kube-system` thing looks suspicious.*

## Accessing namespaces

- By default, `kubectl` uses the `default` namespace
- We can switch to a different namespace with the `-n` option
- List the pods in the `kube-system` namespace:

```
kubectl -n kube-system get pods
```

*Ding ding ding ding ding!*

## What are all these pods?

- `etcd` is our etcd server
- `kube-apiserver` is the API server
- `kube-controller-manager` and `kube-scheduler` are other master components
- `kube-dns` is an additional component (not mandatory but super useful, so it's there)
- `kube-proxy` is the (per-node) component managing port mappings and such
- `weave` is the (per-node) component managing the network overlay
- the `READY` column indicates the number of containers in each pod
- the pods with a name ending with `-node1` are the master components (they have been specifically "pinned" to the master node)

# Running our first containers on Kubernetes

- First things first: we cannot run a container
- We are going to run a pod, and in that pod there will be a single container
- In that container in the pod, we are going to run a simple ping command
- Then we are going to start additional copies of the pod

## Starting a simple pod with `kubectl run`

- We need to specify at least a name and the image we want to use
- Let's ping `8.8.8.8`, Google's public DNS

```
kubectl run pingpong --image alpine ping 8.8.8.8
```

- OK, what just happened?

# Behind the scenes of `kubectl run`

- Let's look at the resources that were created by `kubectl run`
- List most resource types:

```
kubectl get all
```

We should see the following things:

- `deploy/pingpong` (the *deployment* that we just created)
- `rs/pingpong-xxxx` (a *replica set* created by the deployment)
- `po/pingpong-yyyy` (a *pod* created by the replica set)

# What are these different things?

- A *deployment* is a high-level construct
  - allows scaling, rolling updates, rollbacks
  - multiple deployments can be used together to implement a canary deployment
  - delegates pods management to *replica sets*
- A *replica set* is a low-level construct
  - makes sure that a given number of identical pods are running
  - allows scaling
  - rarely used directly
- A *replication controller* is the (deprecated) predecessor of a replica set

# Our `pingpong` deployment

- `kubectl run` created a *deployment*, `deploy/pingpong`
- That deployment created a *replica set*, `rs/pingpong-xxxx`
- That *replica set* created a *pod*, `po/pingpong-yyyy`
- We'll see later how these folks play together for:
  - scaling
  - high availability
  - rolling updates

# Viewing container output

- Let's use the `kubectl logs` command

- We will pass either a *pod name*, or a *type/name* (E.g. if we specify a deployment or replica set, it will get the first pod in it)
- Unless specified otherwise, it will only show logs of the first container in the pod (Good thing there's only one in ours!)
- View the result of our ping command:

```
kubectl logs deploy/pingpong
```

## Streaming logs in real time

- Just like `docker logs`, `kubectl logs` supports convenient options:
  - `-f/--follow` to stream logs in real time (à la tail `-f`)
  - `--tail` to indicate how many lines you want to see (from the end)
  - `--since` to get logs only after a given timestamp
- View the latest logs of our ping command:

```
kubectl logs deploy/pingpong --tail 1 --follow
```

## Scaling our application

- We can create additional copies of our container (or rather our pod) with `kubectl scale`
- Scale our pingpong deployment:

```
kubectl scale deploy/pingpong --replicas 8
```

Note: what if we tried to scale `rs/pingpong-xxxx`? We could! But the *deployment* would notice it right away, and scale back to the initial level.

## Resilience

- The deployment pingpong watches its replica set
- The replica set ensures that the right number of pods are running
- What happens if pods disappear?
- In a separate window, list pods, and keep watching them:

```
kubectl get pods -w
```

`Ctrl-C` to terminate watching.

- If you wanted to destroy a pod, you would use this pattern where `yyyy` was the identifier of the particular pod:

```
kubectl delete pod pingpong-yyyy
```

## What if we wanted something different?

- What if we wanted to start a “one-shot” container that *doesn't* get restarted?
- We could use `kubectl run --restart=OnFailure` or `kubectl run --restart=Never`
- These commands would create *jobs* or *pods* instead of *deployments*
- Under the hood, `kubectl run` invokes “generators” to create resource descriptions
- We could also write these resource descriptions ourselves (typically in YAML), and create them on the cluster with `kubectl apply -f` (discussed later)
- With `kubectl run --schedule=`…, we can also create *cronjobs*

## Viewing logs of multiple pods

- When we specify a deployment name, only one single pod's logs are shown
- We can view the logs of multiple pods by specifying a *selector*
- A selector is a logic expression using *labels*
- Conveniently, when `you kubectl run somename`, the associated objects have a `run=somename` label
- View the last line of log from all pods with the `run=pingpong` label:

```
kubectl logs -l run=pingpong --tail 1
```

- Unfortunately, `--follow` cannot (yet) be used to stream the logs from multiple containers.

## Clean-up

- Clean up your deployment by deleting `pingpong`

```
kubectl delete deploy/pingpong
```

# Exposing containers

- `kubectl expose` creates a *service* for existing pods
- A *service* is a stable address for a pod (or a bunch of pods)
- If we want to connect to our pod(s), we need to create a *service*
- Once a service is created, `kube-dns` will allow us to resolve it by name (i.e. after creating service `hello`, the name `hello` will resolve to something)
- There are different types of services, detailed on the following slides:

`ClusterIP`, `NodePort`, `LoadBalancer`, `ExternalName`

# Basic service types

- `ClusterIP` (default type)

a virtual IP address is allocated for the service (in an internal, private range) this IP address is reachable only from within the cluster (nodes and pods) our code can connect to the service using the original port number

- `NodePort`

  a port is allocated for the service (by default, in the 30000-32768 range) that port is made available on all our nodes and anybody can connect to it our code must be changed to connect to that new port number

These service types are always available.

Under the hood: `kube-proxy` is using a userland proxy and a bunch of `iptables` rules.

# More service types

- `LoadBalancer`
  - an external load balancer is allocated for the service
  - the load balancer is configured accordingly (e.g.: a `NodePort` service is created, and the load balancer sends traffic to that port)
- `ExternalName`
  - the DNS entry managed by `kube-dns` will just be a `CNAME` to a provided record
  - no port, no IP address, no nothing else is allocated

# Running containers with open ports

- Since ping doesn't have anything to connect to, we'll have to run something else
- Start a bunch of ElasticSearch containers:

```
kubectl run elastic --image=elasticsearch:2 --replicas=4
```

- Watch them being started:

```
kubectl get pods -w
```

The `-w` option "watches" events happening on the specified resources.

Note: please DO NOT call the service `search`. It would collide with the TLD.

# Exposing our deployment

- We'll create a default `ClusterIP` service
- Expose the ElasticSearch HTTP API port:

```
kubectl expose deploy/elastic --port 9200
```

- Look up which IP address was allocated:

```
kubectl get svc
```

# Services are layer 4 constructs

- You can assign IP addresses to services, but they are still *layer 4* (i.e. a service is not an IP address; it's an IP address + protocol + port)
- This is caused by the current implementation of `kube-proxy` (it relies on mechanisms that don't support layer 3)
- As a result: *you have to* indicate the port number for your service
- Running services with arbitrary port (or port ranges) requires hacks (e.g. host networking mode)

## Testing our service

- We will now send a few HTTP requests to our ElasticSearch pods
- Let's obtain the IP address that was allocated for our service, *programatically*:

```
IP=$(kubectl get svc elastic -o go-template --template '{{ .spec.clusterIP
}}')
```

- Send a few requests:

```
curl http://$IP:9200/
```

Our requests are load balanced across multiple pods.

## Clean up

- We're done with the `elastic` deployment, so let's clean it up

```
kubectl delete deploy/elastic
```

# Our app on Kube

## What's on the menu?

In this part, we will:

- **build** images for our app,
- **ship** these images with a registry,
- **run** deployments using these images,
- expose these deployments so they can communicate with each other,
- expose the web UI so we can access it from outside.

## The plan

- Build on our control node (`node1`)
- Tag images so that they are named `$USERNAME/servicename`
- Upload them to a Docker Hub

- Create deployments using the images
- Expose (with a `ClusterIP`) the services that need to communicate
- Expose (with a `NodePort`) the WebUI

# Setup

- In the first terminal, set an environment variable for your Docker Hub user name. It can be the same Docker Hub user name that you used to log in to the terminals on this site.

```
export USERNAME=YourUserName
```

- Make sure you're still in the `dockercoins` directory.

```
pwd
```

# A note on registries

- For this workshop, we'll use Docker Hub. There are a number of other options, including two provided by Docker.
- Docker also provides:
  - Docker Trusted Registry which adds in a lot of security and deployment features including security scanning, and role-based access control.
  - Docker Open Source Registry.

# Docker Hub

- Docker Hub is the default registry for Docker.
  - Image names on Hub are just `$USERNAME/$IMAGENAME` or `$ORGANIZATIONNAME/$IMAGENAME`.
  - Official images can be referred to as just `$IMAGENAME`.
  - To use Hub, make sure you have an account. Then type `docker login` in the terminal and login with your username and password.
- Using Docker Trusted Registry, Docker Open Source Registry is very similar.
  - Image names on other registries are `$REGISTRYPATH/$USERNAME/$IMAGENAME` or `$REGISTRYPATH/$ORGANIZATIONNAME/$IMAGENAME`.
  - Login using `docker login $REGISTRYPATH`.

# Building and pushing our images

- We are going to use a convenient feature of Docker Compose
- Go to the `stacks` directory:

```
cd ~/dockercoins/stacks
```

- Build and push the images:

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

Just in case you were wondering … Docker "services" are not Kubernetes "services".

## Deploying all the things

- We can now deploy our code (as well as a redis instance)
- Deploy `redis`:

```
kubectl run redis --image=redis
```

- Deploy everything else:

```
for SERVICE in hasher rng webui worker; do
  kubectl run $SERVICE --image=$USERNAME/$SERVICE -l app=$SERVICE
done
```

## Is this working?

- After waiting for the deployment to complete, let's look at the logs!
- (Hint: use `kubectl get deploy -w` to watch deployment events)
- Look at some logs:

```
kubectl logs deploy/rng
kubectl logs deploy/worker
```

🤔 `rng` is fine … But not `worker`.

💡 Oh right! We forgot to `expose`.

# Exposing services

# Exposing services internally

- Three deployments need to be reachable by others: `hasher`, `redis`, `rng`
- `worker` doesn't need to be exposed
- `webui` will be dealt with later
- Expose each deployment, specifying the right port:

```
kubectl expose deployment redis --port 6379
kubectl expose deployment rng --port 80
kubectl expose deployment hasher --port 80
```

# Is this working yet?

- The `worker` has an infinite loop, that retries 10 seconds after an error
- Stream the worker's logs:

```
kubectl logs deploy/worker --follow
```

(Give it about 10 seconds to recover)

- We should now see the `worker`, well, working happily.

# Exposing services for external access

- Now we would like to access the Web UI
- We will expose it with a `NodePort` (just like we did for the registry)
- Create a `NodePort` service for the Web UI:

```
kubectl create service nodeport webui --tcp=80 --node-port=30001
```

- Check the port that was allocated:

```
kubectl get svc
```

## Accessing the web UI

- We can now connect to *any node*, on the allocated node port, to view the web UI

Click on this link

*Alright, we're back to where we started, when we were running on a single node!*

# Security implications of `kubectl apply`

- When we do `kubectl apply -f <URL>`, we create arbitrary resources
- Resources can be evil; imagine a `deployment` that …
  - starts bitcoin miners on the whole cluster
  - hides in a non-default namespace
  - bind-mounts our nodes' filesystem
  - inserts SSH keys in the root account (on the node)
  - encrypts our data and ransoms it
  - ☠☠☠

## `kubectl apply` is the new `curl | sh`

- `curl | sh` is convenient
- It's safe if you use HTTPS URLs from trusted sources
- `kubectl apply -f` is convenient
- It's safe if you use HTTPS URLs from trusted sources
- It introduces new failure modes
- Example: the official setup instructions for most pod networks

# Scaling a deployment

- We will start with an easy one: the `worker` deployment

```
kubectl get pods
kubectl get deployments
```

- Now, create more `worker` replicas:

```
kubectl scale deploy/worker --replicas=10
```

- After a few seconds, the graph in the web UI should show up. (And peak at 10 hashes/second, just like when we were running on a single one.)

# Daemon sets

- What if we want one (and exactly one) instance of rng per node?
- If we just scale deploy/rng to 2, nothing guarantees that they spread
- Instead of a deployment, we will use a daemonset
- Daemon sets are great for cluster-wide, per-node processes:
  - kube-proxy
  - weave (our overlay network)
  - monitoring agents
  - hardware management tools (e.g. SCSI/FC HBA agents)
  - etc.
- They can also be restricted to run only on some nodes.

# Creating a daemon set

- Unfortunately, as of Kubernetes 1.9, the CLI cannot create daemon sets
- More precisely: it doesn't have a subcommand to create a daemon set
- But any kind of resource can always be created by providing a YAML description:

```
kubectl apply -f foo.yaml
```

- How do we create the YAML file for our daemon set?
  - option 1: read the docs
  - option 2: `vi` our way out of it

# Creating the YAML file for our daemon set

- Let's start with the YAML file for the current `rng` resource
- Dump the rng resource in YAML:

```
kubectl get deploy/rng -o yaml --export >rng.yml
```

Edit `rng.yml`

Note: `--export` will remove "cluster-specific" information, i.e.:

namespace (so that the resource is not tied to a specific namespace) status and creation timestamp (useless when creating a new resource) resourceVersion and uid (these would cause… *interesting* problems)

# "Casting" a resource to another

- What if we just changed the `kind` field?
- (It can't be that easy, right?)

Change `kind: Deployment` to `kind: DaemonSet`

Save, quit

Try to create our new resource:

```
kubectl apply -f rng.yml
```

- We all knew this couldn't be that easy, right!

# Understanding the problem

- The core of the error is:

```
error validating data:
[ValidationError(DaemonSet.spec):
unknown field "replicas" in io.k8s.api.extensions.v1beta1.DaemonSetSpec,
...
```

- *Obviously*, it doesn't make sense to specify a number of replicas for a daemon set
- Workaround: fix the YAML
  - remove the `replicas` field
  - remove the `strategy` field (which defines the rollout mechanism for a deployment)
  - remove the `status: {}` line at the end
- Or, we could also …

# Use the `--force`, Luke

- We could also tell Kubernetes to ignore these errors and try anyway
- The `--force` flag actual name is `--validate=false`
- Try to load our YAML file and ignore errors:

```
kubectl apply -f rng.yml --validate=false
```

Use the –force, Luke We could also tell Kubernetes to ignore these errors and try anyway

The –force flag actual name is –validate=false

Try to load our YAML file and ignore errors: kubectl apply -f rng.yml –validate=false

Wait … Now, *can* it be that easy?

# Checking what we've done

- Did we transform our `deployment` into a `daemonset`?
- Look at the resources that we have now:

```
kubectl get all
```

We have both `deploy/rng` and `ds/rng` now!

And one too many pods…

# Explanation

- You can have different resource types with the same name (i.e. a *deployment* and a *daemonset* both named `rng`)
- We still have the old `rng` *deployment*
- But now we have the new `rng` *daemonset* as well
- If we look at the pods, we have:
  - *one pod* for the deployment
  - *one pod per node* for the daemonset

# What are all these pods doing?

- Let's check the logs of all these `rng` pods
- All these pods have a `run=rng` label:
  - the first pod, because that's what `kubectl run` does
  - the other ones (in the daemon set), because we *copied the spec from the first one*
- Therefore, we can query everybody's logs using that `run=rng` selector
- Check the logs of all the pods having a label `run=rng`:

```
kubectl logs -l run=rng --tail 1
```

- It appears that *all the pods* are serving requests at the moment.

# Removing the first pod from the load balancer

- What would happen if we removed that pod, with `kubectl delete pod ...`?

  The replicaset would re-create it immediately.
- What would happen if we removed the `run=rng` label from that pod?

  The `replicaset` would re-create it immediately.

  … Because what matters to the `replicaset` is the number of pods *matching that selector*.
- But but but … Don't we have more than one pod with `run=rng` now?

  The answer lies in the exact selector used by the `replicaset` …

# Deep dive into selectors

- Let's look at the selectors for the `rng` *deployment* and the associated *replica set*
- Show detailed information about the `rng` deployment:

```
kubectl describe deploy rng
```

- Show detailed information about the `rng` replica:

```
kubectl describe rs rng-yyyy
```

- The replica set selector also has a `pod-template-hash`, unlike the pods in our daemon set.

# Updating a service through labels and selectors

- What if we want to drop the `rng` deployment from the load balancer?
- Option 1:
  - destroy it
- Option 2:
  - add an extra *label* to the daemon set
  - update the service *selector* to refer to that *label*

Of course, option 2 offers more learning opportunities. Right?

## Add an extra label to the daemon set

- We will update the daemon set "spec"
- Option 1:
  - edit the `rng.yml` file that we used earlier
  - load the new definition with `kubectl apply`
- Option 2:
  - use `kubectl edit`

*If you feel like you got this, feel free to try directly. We've included a few hints on the next slides for your convenience!*

## We've put resources in your resources

- Reminder: a daemon set is a resource that creates more resources!
- There is a difference between:
  - the label(s) of a resource (in the `metadata` block in the beginning)
  - the selector of a resource (in the `spec` block)
  - the label(s) of the resource(s) created by the first resource (in the `template` block)
- You need to update the selector and the template (metadata labels are not mandatory)
- The template must match the selector (i.e. the resource will refuse to create resources that it will not select)

## Adding our label

Let's add a label `isactive: yes`

In YAML, yes should be quoted; i.e. `isactive: "yes"`

- Update the daemon set to add `isactive: "yes"` to the selector and template label:

```
kubectl edit daemonset rng
```

```
spec:
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: rng
      isactive: "yes"
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: rng
        isactive: "yes"
```

- Update the service to add `isactive: "yes"` to its selector:

```
kubectl edit service rng
```

## Checking what we've done

- Check the logs of all `run=rng` pods to confirm that only 2 of them are now active:

```
kubectl logs -l run=rng
```

- The timestamps should give us a hint about how many pods are currently receiving traffic.
- Look at the pods that we have right now:

```
kubectl get pods
```

## More labels, more selectors, more problems?

- Bonus exercise 1: clean up the pods of the "old" daemon set
- Bonus exercise 2: how could we have done this to avoid creating new pods?

# Rolling updates

- By default (without rolling updates), when a scaled resource is updated:
  - new pods are created
  - old pods are terminated
  - … all at the same time
  - if something goes wrong, ¯_(ツ)_/¯
- With rolling updates, when a resource is updated, it happens progressively
- Two parameters determine the pace of the rollout: `maxUnavailable` and `maxSurge`

- They can be specified in absolute number of pods, or percentage of the `replicas` count
- At any given time …
    - there will always be at least `replicas`-`maxUnavailable` pods available
    - there will never be more than `replicas`+`maxSurge` pods in total
    - there will therefore be up to `maxUnavailable`+`maxSurge` pods being updated
- We have the possibility to rollback to the previous version (if the update fails or is unsatisfactory in any way)

# Rolling updates in practice

- As of Kubernetes 1.8, we can do rolling updates with:

  `deployments`, `daemonsets`, `statefulsets`
- Editing one of these resources will automatically result in a rolling update
- Rolling updates can be monitored with the `kubectl rollout` subcommand

# Building a new version of the `worker` service

- Edit `dockercoins/worker/worker.py`, update the sleep line to sleep 1 second
- Go to the stack directory:

```
cd stacks
```

- Build a new tag and push it to the registry:

```
export TAG=v0.2
 docker-compose -f dockercoins.yml build
 docker-compose -f dockercoins.yml push
```

# Rolling out the new worker service

- Let's monitor what's going on by opening a few terminals, and run:

```
 kubectl get pods -w
 kubectl get replicasets -w
 kubectl get deployments -w
```

- Update worker either with kubectl edit, or by running:

```
kubectl set image deploy worker worker=$USERNAME/worker:$TAG
```

- That rollout should be pretty quick. What shows in the web UI?

# Rolling out an error

- What happens if we make a mistake?

- Update worker by specifying a non-existent image:

```
export TAG=v0.3
kubectl set image deploy worker worker=$REGISTRY/worker:$TAG
```

- Check what's going on:

```
kubectl rollout status deploy worker
```

- Our rollout is stuck. However, the app is not dead (just 10% slower).

# Recovering from a bad rollout

- We could push some v0.3 image (the pod retry logic will eventually catch it and the rollout will proceed)
- Or we could invoke a manual rollback
- Cancel the deployment and wait for the dust to settle down:

```
kubectl rollout undo deploy worker
kubectl rollout status deploy worker
```

# Changing rollout parameters

- We want to:
  - revert to `v0.1` (which we now realize we didn't tag - yikes!)
  - be conservative on availability (always have desired number of available workers)
  - be aggressive on rollout speed (update more than one pod at a time)
  - give some time to our workers to "warm up" before starting more
- The corresponding changes can be expressed in the following YAML snippet:

```
spec:
  template:
    spec:
      containers:
      - name: worker
        image: $USERNAME/worker:latest
  strategy:
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 3
  minReadySeconds: 10
```

# Applying changes through a YAML patch

- We could use `kubectl edit deployment worker`
- But we could also use `kubectl patch` with the exact YAML shown before
- Apply all our changes and wait for them to take effect:

```
kubectl patch deployment worker -p "
spec:
  template:
    spec:
      containers:
      - name: worker
        image: $USERNAME/worker:latest
  strategy:
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 3
  minReadySeconds: 10
"
kubectl rollout status deployment worker
```

# Next steps

*Alright, how do I get started and containerize my apps?*

Suggested containerization checklist:

- write a Dockerfile for one service in one app
- write Dockerfiles for the other (buildable) services
- write a Compose file for that whole app
- make sure that devs are empowered to run the app in containers
- set up automated builds of container images from the code repo
- set up a CI pipeline using these container images
- set up a CD pipeline (for staging/QA) using these images

And *then* it is time to look at orchestration!

# Namespaces

- Namespaces let you run multiple identical stacks side by side
- Two namespaces (*e.g.* `blue` and `green`) can each have their own `redis` service
- Each of the two `redis` services has its own `ClusterIP`
- `kube-dns` creates two entries, mapping to these two `ClusterIP` addresses:
- `redis.blue.svc.cluster.local` and `redis.green.svc.cluster.local`
- Pods in the `blue` namespace get a *search suffix* of `blue.svc.cluster.local`

- As a result, resolving `redis` from a pod in the `blue` namespace yields the “local” `redis`

This does not provide *isolation*! That would be the job of network policies.

# Stateful services (databases etc.)

- As a first step, it is wiser to keep stateful services *outside* of the cluster
- Exposing them to pods can be done with multiple solutions:
- `ExternalName` services (`redis.blue.svc.cluster.local` will be a `CNAME` record)
- `ClusterIP` services with explicit `Endpoints` (instead of letting Kubernetes generate the endpoints from a selector)
- Ambassador services (application-level proxies that can provide credentials injection and more)

# Stateful services (second take)

- If you really want to host stateful services on Kubernetes, you can look into:
  - volumes (to carry persistent data)
  - storage plugins
  - persistent volume claims (to ask for specific volume characteristics)
  - stateful sets (pods that are *not* ephemeral)

# HTTP traffic handling

- *Services* are layer 4 constructs
- HTTP is a layer 7 protocol
- It is handled by *ingresses* (a different resource kind)
- Ingresses allow:
  - virtual host routing
  - session stickiness
  - URI mapping
  - and much more!

# Logging and metrics

- Logging is delegated to the container engine
- Metrics are typically handled with Prometheus

# Managing the configuration of our applications

- Two constructs are particularly useful: secrets and config maps
- They allow to expose arbitrary information to our containers
- **Avoid** storing configuration in container images (There are some exceptions to that rule, but it’s generally a Bad Idea)
- **Never** store sensitive information in container images (It’s the container equivalent of the password on a post-it note on your screen)

# Cluster federation

- Kubernetes master operation relies on etcd
- etcd uses the Raft protocol
- Raft recommends low latency between nodes
- What if our cluster spreads to multiple regions?
- Break it down in local clusters
- Regroup them in a cluster *federation*
- Synchronize resources across clusters
- Discover resources across clusters