Kubernetes bootcamp: Deploying and scaling microservices

WiFi: OReilly18
Password: velocity

Be kind to the WiFi!
Don't use your hotspot.
Don't stream videos or download big files during the workshop.
Thank you!

Slides: http://k8s2d.container.training/

Intros

- Hello! I'm Jérôme (@jpetazzo, Enix SAS)
- The workshop will run from 9am to 5pm
- Lunch will be served at 12:30pm (in Rhinelander)
- Morning and afternoon breaks are scheduled at 10:30am and 3pm (in Sutton Foyer)
- Feel free to interrupt for questions at any time
- *Especially when you see full screen container pictures!*
- Live feedback, questions, help: Gitter

A brief introduction

- This was initially written by Jérôme Petazzoni to support in-person, instructor-led workshops and tutorials
- Credit is also due to multiple contributors — thank you!
- 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 Kubernetes documentation ...
- ... And looking for answers on StackOverflow and other outlets

About these slides

- All the content is available in a public GitHub repository:

  https://github.com/jpetazzo/container.training

- You can get updated "builds" of the slides there:

  http://container.training/

About these slides

- All the content is available in a public GitHub repository:

  https://github.com/jpetazzo/container.training

- You can get updated "builds" of the slides there:

  http://container.training/

- Typos? Mistakes? Questions? Feel free to hover over the bottom of the slide ...

👇 Try it! The source file will be shown and you can view it on GitHub and fork and edit it.

Extra details

- This slide has a little magnifying glass in the top left corner
- This magnifying glass indicates slides that provide extra details
- Feel free to skip them if:
  - you are in a hurry
  - you are new to this and want to avoid cognitive overload
  - you want only the most essential information
- You can review these slides another time if you want, they'll be waiting for you ☺

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Chapter 9

Pre-requirements

Pre-requirements

- Be comfortable with the UNIX command line
  - navigating directories
  - editing files
  - a little bit of bash-fu (environment variables, loops)
- Some Docker knowledge
  - `docker run`, `docker ps`, `docker build`
  - ideally, you know how to write a Dockerfile and build it (even if it's a `FROM` line and a couple of `RUN` commands)
- It's totally OK if you are not a Docker expert!

Tell me and I forget.
Teach me and I remember.
Involve me and I learn.

Misattributed to Benjamin Franklin

(Probably inspired by Chinese Confucian philosopher Xunzi)

Hands-on sections

- The whole workshop is hands-on
- We are going to build, ship, and run containers!
- 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 http://k8s2d.container.training/ to view these slides
- Join the chat room: Gitter

Where are we going to run our containers?


YOU GET A CLUSTER AND YOU GET A CLUSTER
EVERYBODY GETS A CLUSTER
imgflip.com

You get a cluster of cloud VMs

- Each person gets a private cluster of cloud VMs (not shared with anybody else)
- They'll remain up for the duration of the workshop
- You should have a little card with login+password+IP addresses
- You can automatically SSH from one VM to another
- The nodes have aliases: `node1`, `node2`, etc.

Why don't we run containers locally?

- Installing that stuff can be hard on some machines

  (32 bits CPU or OS... Laptops without administrator access... etc.)

- *"The whole team downloaded all these container images from the WiFi!
  ... and it went great!"* (Literally no-one ever)

- All you need is a computer (or even a phone or tablet!), with:

  - an internet connection

  - a web browser

  - an SSH client

SSH clients

- On Linux, OS X, FreeBSD... you are probably all set
- On Windows, get one of these:
  - putty
  - Microsoft Win32 OpenSSH
  - Git BASH
  - MobaXterm
- On Android, JuiceSSH (Play Store) works pretty well
- Nice-to-have: Mosh instead of SSH, if your internet connection tends to lose packets

What is this Mosh thing?

You don't have to use Mosh or even know about it to follow along.
We're just telling you about it because some of us think it's cool!

- Mosh is "the mobile shell"
- It is essentially SSH over UDP, with roaming features
- It retransmits packets quickly, so it works great even on lossy connections

  (Like hotel or conference WiFi)
- It has intelligent local echo, so it works great even in high-latency connections

  (Like hotel or conference WiFi)
- It supports transparent roaming when your client IP address changes

  (Like when you hop from hotel to conference WiFi)

Using Mosh

- To install it: `(apt|yum|brew) install mosh`
- It has been pre-installed on the VMs that we are using
- To connect to a remote machine: `mosh user@host`

  (It is going to establish an SSH connection, then hand off to UDP)
- It requires UDP ports to be open

  (By default, it uses a UDP port between 60000 and 61000)

Connecting to our lab environment

Exercise

- Log into the first VM (`node1`) with your SSH client

- Check that you can SSH (without password) to `node2`:

```
ssh node2
```

- Type `exit` or `^D` to come back to `node1`

If anything goes wrong — ask for help!

Doing or re-doing the workshop on your own?

- Use something like Play-With-Docker or Play-With-Kubernetes

  Zero setup effort; but environment are short-lived and might have limited resources

- Create your own cluster (local or cloud VMs)

  Small setup effort; small cost; flexible environments

- Create a bunch of clusters for you and your friends (instructions)

  Bigger setup effort; ideal for group training

We will (mostly) interact with node1 only

These remarks apply only when using multiple nodes, of course.

- Unless instructed, **all commands must be run from the first VM, `node1`**
- We will only checkout/copy the code on `node1`
- During normal operations, we do not need access to the other nodes
- If we had to troubleshoot issues, we would use a combination of:
  - SSH (to access system logs, daemon status...)
  - Docker API (to check running containers and container engine status)

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

Tmux is a terminal multiplexer like `screen`.

You don't have to use it or even know about it to follow along.
But some of us like to use it to switch between terminals.
It has been preinstalled on your workshop nodes.

- 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

Versions installed

- Kubernetes 1.12.0
- Docker Engine 18.06.1-ce
- Docker Compose 1.21.1

Exercise

- Check all installed versions:

```
kubectl version
docker version
docker-compose -v
```

Kubernetes and Docker compatibility

- Kubernetes 1.12.x only validates Docker Engine versions 1.11.2 to 1.13.1 and 17.03.x

Kubernetes and Docker compatibility

- Kubernetes 1.12.x only validates Docker Engine versions 1.11.2 to 1.13.1 and 17.03.x
- Are we living dangerously?

Kubernetes and Docker compatibility

- Kubernetes 1.12.x only validates Docker Engine versions 1.11.2 to 1.13.1 and 17.03.x

- Are we living dangerously?

- "Validates" = continuous integration builds

- The Docker API is versioned, and offers strong backward-compatibility

  (If a client uses e.g. API v1.25, the Docker Engine will keep behaving the same way)


한진 보스톤
HANJIN BOSTON
MAJURO

Our sample application

Previous section | Back to table of contents | Next section

Our sample application

- We will clone the GitHub repository onto our `node1`
- The repository also contains scripts and tools that we will use through the workshop

Exercise

- Clone the repository on `node1`:

```
git clone https://github.com/jpetazzo/container.training
```

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

Downloading and running the application

Let's start this before we look around, as downloading will take a little time...

Exercise

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

```
cd ~/container.training/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.

More detail on our sample application

- Visit the GitHub repository with all the materials of this workshop:
  https://github.com/jpetazzo/container.training

- 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

Service discovery in container-land

- We do not hard-code IP addresses in the code
- We do not hard-code FQDN in the code, either
- We just connect to a service name, and container-magic does the rest

  (And by container-magic, we mean "a crafty, dynamic, embedded DNS server")

Example in `worker/worker.py`

```python
redis = Redis("redis")


def get_random_bytes():
    r = requests.get("http://rng/32")
    return r.content


def hash_bytes(data):
    r = requests.post("http://hasher/",
                      data=data,
                      headers={"Content-Type": "application/octet-stream"})
```

(Full source code available here)

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
- Network aliases are automatically namespaced
  - you can have multiple apps declaring and using a service named `database`
  - containers in the blue app will resolve `database` to the IP of the blue database
  - containers in the green app will resolve `database` to the IP of the green database

What's this application?

What's this application?

- It is a DockerCoin miner! 💰🐳📦🚢

What's this application?

- It is a DockerCoin miner! 💰🐳📦🚢

- No, you can't buy coffee with DockerCoins

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

Our application at work

- On the left-hand side, the "rainbow strip" shows the container names
- On the right-hand side, we see the output of our containers
- We can see the `worker` service making requests to `rng` and `hasher`
- For `rng` and `hasher`, we see HTTP access logs

Connecting to the web UI

- "Logs are exciting and fun!" (No-one, ever)
- 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

A drawing area should show up, and after a few seconds, a blue graph will appear.

Why does the speed seem irregular?

- It *looks like* the speed is approximately 4 hashes/second
- Or more precisely: 4 hashes/second, with regular dips down to zero
- Why?

Why does the speed seem irregular?

- It *looks like* the speed is approximately 4 hashes/second

- Or more precisely: 4 hashes/second, with regular dips down to zero

- Why?

- The app actually has a constant, steady speed: 3.33 hashes/second (which corresponds to 1 hash every 0.3 seconds, for *reasons*)

- Yes, and?

The reason why this graph is *not awesome*

- 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 - 0 etc.
- What can we conclude from this?

The reason why this graph is *not awesome*

- 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 - 0 etc.
- What can we conclude from this?
- "I'm clearly incapable of writing good frontend code!" 😀 — Jérôme

Stopping the application

- If we interrupt Compose (with `^C`), it will politely ask the Docker Engine to stop the app
- The Docker Engine will send a `TERM` signal to the containers
- If the containers do not exit in a timely manner, the Engine sends a `KILL` signal

Exercise

- Stop the application by hitting `^C`

Stopping the application

- If we interrupt Compose (with `^C`), it will politely ask the Docker Engine to stop the app
- The Docker Engine will send a `TERM` signal to the containers
- If the containers do not exit in a timely manner, the Engine sends a `KILL` signal

Exercise

- Stop the application by hitting `^C`

Some containers exit immediately, others take longer.

The containers that do not handle `SIGTERM` end up being killed after a 10s timeout. If we are very impatient, we can hit `^C` a second time!

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.

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 1` 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 up -d --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 up -d --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

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 3 localhost:8001
```

- Check the latency of `hasher`:

```
httping -c 3 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
```


© Lode Van de Velde

Kubernetes concepts

Kubernetes concepts

- Kubernetes is a container management system

- It runs and manages containerized applications on a cluster

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

Basic things we can ask Kubernetes to do

- Start 5 containers using image `atseashop/api:v1.3`

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

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`

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

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

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`

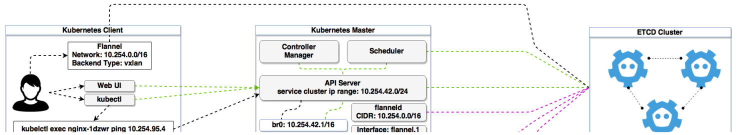
Basic things we can ask Kubernetes to do

- 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 Client
Flannel
Network: 10.254.0.0/16
Backend Type: vxlan
Web UI
kubectl
kubelctl exec nginx-1dzwr ping 10.254.95.4
Kubernetes Master
Controller Manager
Scheduler
API Server
service cluster ip range: 10.254.42.0/24
br0: 10.254.42.1/16
flanneld
CIDR: 10.254.0.0/16
Interface: flannel.1
ETCD Cluster

Kubernetes architecture

- Ha ha ha ha

- OK, I was trying to scare you, it's much simpler than that ❤️

Credits

- The first schema is a Kubernetes cluster with storage backed by multi-path iSCSI

  (Courtesy of Yongbok Kim)

- The second one is a simplified representation of a Kubernetes cluster

  (Courtesy of Imesh Gunaratne)

Kubernetes architecture: the nodes

- The nodes executing our containers run a 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"

  (You might see that word in older articles or documentation)

Kubernetes architecture: the control plane

- 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 the control plane of our cluster
- The control plane is also called the "master"

Running the control plane on special nodes

- It is common to reserve a dedicated node for the control plane

  (Except for single-node development clusters, like when using minikube)

- This node is then called a "master"

  (Yes, this is ambiguous: is the "master" a node, or the whole control plane?)

- Normal applications are restricted from running on this node

  (By using a mechanism called "taints")

- When high availability is required, each service of the control plane must be resilient

- The control plane is then replicated on multiple nodes

  (This is sometimes called a "multi-master" setup)

Running the control plane outside containers

- The services of the control plane can run in or out of containers

- For instance: since `etcd` is a critical service, some people deploy it directly on a dedicated cluster (without containers)

  (This is illustrated on the first "super complicated" schema)

- In some hosted Kubernetes offerings (e.g. AKS, GKE, EKS), the control plane is invisible

  (We only "see" a Kubernetes API endpoint)

- In that case, there is no "master node"

For this reason, it is more accurate to say "control plane" rather than "master".

Do we need to run Docker at all?

No!

Do we need to run Docker at all?

No!

- By default, Kubernetes uses the Docker Engine to run containers

- We could also use `rkt` ("Rocket") from CoreOS

- Or leverage other pluggable runtimes through the *Container Runtime Interface*

  (like CRI-O, or containerd)

Do we need to run Docker at all?

Yes!

Do we need to run Docker at all?

Yes!

- In this workshop, we run our app on a single node first
- We will need to build images and ship them around
- We can do these things without Docker
  (and get diagnosed with NIH[1] syndrome)
- Docker is still the most stable container engine today
  (but other options are maturing very quickly)

[1] Not Invented Here

Do we need to run Docker at all?

- On our development environments, CI pipelines ... :

  Yes, almost certainly

- On our production servers:

  Yes (today)

  Probably not (in the future)

More information about CRI on the Kubernetes blog

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:

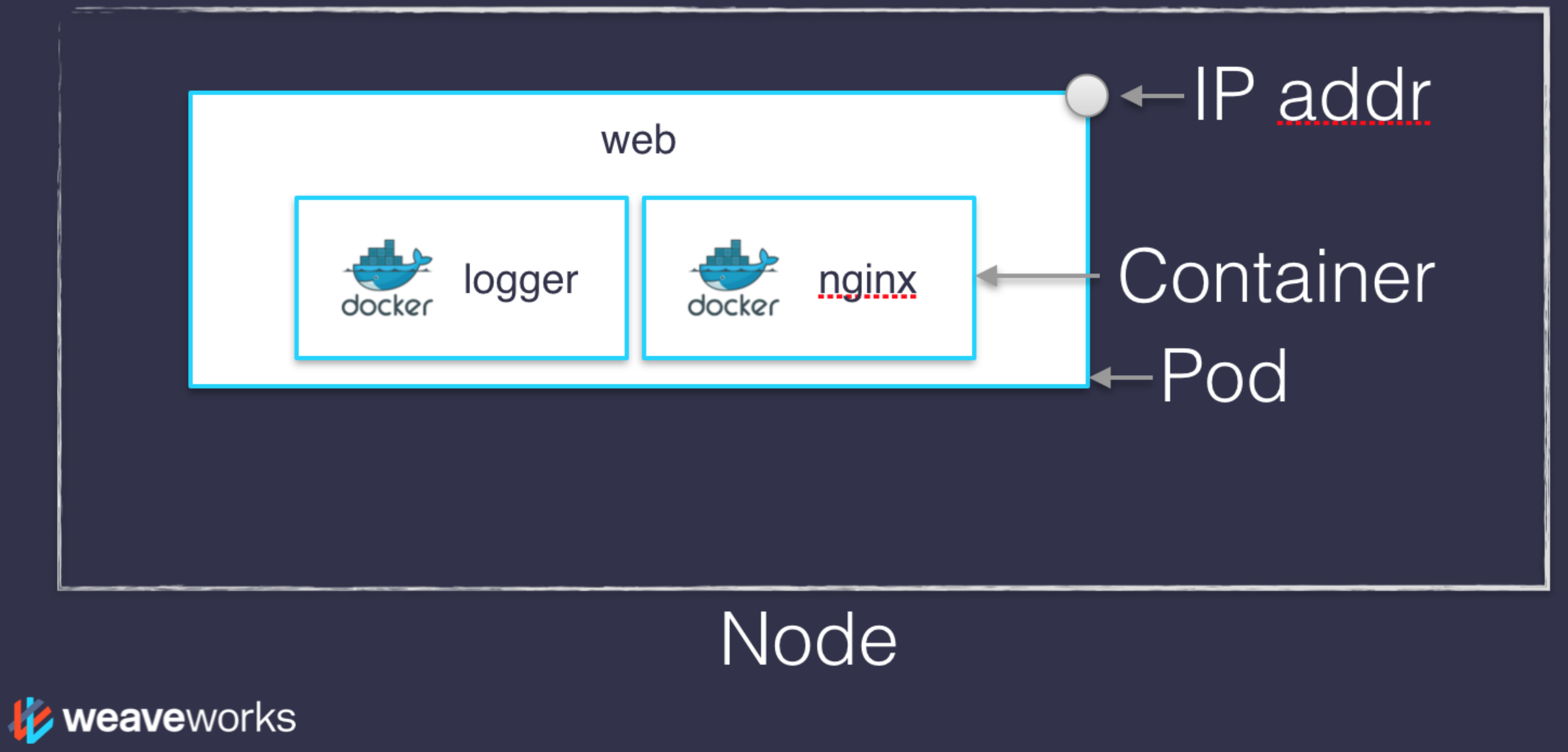
  - node (a machine — physical or virtual — in our cluster)
  - pod (group of containers running together on a node)
  - service (stable network endpoint to connect to one or multiple containers)
  - namespace (more-or-less isolated group of things)
  - secret (bundle of sensitive data to be passed to a container)

  And much more!

- We can see the full list by running `kubectl api-resources`

  (In Kubernetes 1.10 and prior, the command to list API resources was `kubectl get`)

Concepts


IP addr
web
docker
logger
docker
nginx
Container
Pod
Node
weaveworks

Credits

- The first diagram is courtesy of Weave Works
  - a *pod* can have multiple containers working together
  - IP addresses are associated with *pods*, not with individual containers
- The second diagram is courtesy of Lucas Käldström, in this presentation
  - it's one of the best Kubernetes architecture diagrams available!

Both diagrams used with permission.

Declarative vs imperative

Previous section | Back to table of contents | Next section

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 a cup.

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 a cup.

- Declarative seems simpler at first ...

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 a cup.

- Declarative seems simpler at first …
- … As long as you know how to brew tea

Declarative vs imperative

- What declarative would really be:

  I want a cup of tea, obtained by pouring an infusion[1] of tea leaves in a cup.

Declarative vs imperative

- What declarative would really be:

  I want a cup of tea, obtained by pouring an infusion[1] of tea leaves in a cup.

  [1]An infusion is obtained by letting the object steep a few minutes in hot[2] water.

Declarative vs imperative

- What declarative would really be:

  I want a cup of tea, obtained by pouring an infusion[1] of tea leaves in a cup.

  [1]An infusion is obtained by letting the object steep a few minutes in hot[2] water.

  [2]Hot liquid is obtained by pouring it in an appropriate container[3] and setting it on a stove.

Declarative vs imperative

- What declarative would really be:

  I want a cup of tea, obtained by pouring an infusion[1] of tea leaves in a cup.

  [1]An infusion is obtained by letting the object steep a few minutes in hot[2] water.

  [2]Hot liquid is obtained by pouring it in an appropriate container[3] and setting it on a stove.

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

Declarative vs imperative

- What declarative would really be:

  I want a cup of tea, obtained by pouring an infusion[1] of tea leaves in a cup.

  [1]An infusion is obtained by letting the object steep a few minutes in hot[2] water.

  [2]Hot liquid is obtained by pouring it in an appropriate container[3] and setting it on a stove.

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

Did you know there was an ISO standard specifying how to brew tea?

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

Previous section | Back to table of contents | Next section

Kubernetes network model

- TL,DR:

  Our cluster (nodes and pods) is one big flat IP network.

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)
- Pods have level 3 (IP) connectivity, but *services* are level 4

  (Services map to a single UDP or TCP port; no 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)

- If necessary, there are alternatives to `kube-proxy`; e.g. `kube-router`

The Container Network Interface (CNI)

- The CNI has a well-defined specification for network plugins
- When a pod is created, Kubernetes delegates the network setup to CNI plugins
- Typically, a CNI plugin will:
  - allocate an IP address (by calling an IPAM plugin)
  - add a network interface into the pod's network namespace
  - configure the interface as well as required routes etc.
- Using multiple plugins can be done with "meta-plugins" like CNI-Genie or Multus
- Not all CNI plugins are equal

  (e.g. they don't all implement network policies, which are required to isolate pods)

First contact with `kubectl`

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)
- On our machines, there is a `~/.kube/config` file with:
  - the Kubernetes API address
  - the path to our TLS certificates used to authenticate
- 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`!

Exercise

- 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

Exercise

- 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

Exercise

- 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 api-resources`
  (In Kubernetes 1.10 and prior, this command used to be `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")

Exercise

- List the services on our cluster with one of these commands:

```
kubectl get services
kubectl get svc
```

Services

- A *service* is a stable endpoint to connect to "something"

  (In the initial proposal, they were called "portals")

Exercise

- List the services on our cluster with one of these commands:

```
kubectl get services
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

Exercise

- Try to connect to the API:

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

  - `-k` is used to skip certificate verification
  - Make sure to replace 10.96.0.1 with the CLUSTER-IP shown by `kubectl get svc`

ClusterIP services

- A `ClusterIP` service is internal, available from the cluster only

- This is useful for introspection from within containers

Exercise

- Try to connect to the API:

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

  - `-k` is used to skip certificate verification

  - Make sure to replace 10.96.0.1 with the CLUSTER-IP shown by `kubectl get svc`

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)

Exercise

- List pods on our cluster:

```
kubectl get pods
```

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)

Exercise

- 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

Exercise

- List the namespaces on our cluster with one of these commands:

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

Namespaces

- Namespaces allow us to segregate resources

Exercise

- List the namespaces on our cluster with one of these commands:

```
kubectl get namespaces
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

Exercise

- List the pods in the `kube-system` namespace:

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

Accessing namespaces

- By default, `kubectl` uses the `default` namespace
- We can switch to a different namespace with the `-n` option

Exercise

- List the pods in the `kube-system` namespace:

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

Ding ding ding ding ding!

The `kube-system` namespace is used for the control plane.

What are all these control plane pods?

- `etcd` is our etcd server
- `kube-apiserver` is the API server
- `kube-controller-manager` and `kube-scheduler` are other master components
- `coredns` provides DNS-based service discovery (replacing kube-dns as of 1.11)
- `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)

What about `kube-public`?

Exercise

- List the pods in the `kube-public` namespace:

```
kubectl -n kube-public get pods
```

What about `kube-public`?

Exercise

- List the pods in the `kube-public` namespace:

```
kubectl -n kube-public get pods
```

- Maybe it doesn't have pods, but what secrets is `kube-public` keeping?

What about `kube-public`?

⌨ Exercise

- List the pods in the `kube-public` namespace:

```
kubectl -n kube-public get pods
```

- Maybe it doesn't have pods, but what secrets is `kube-public` keeping?

⌨ Exercise

- List the secrets in the `kube-public` namespace:

```
kubectl -n kube-public get secrets
```

What about `kube-public`?

Exercise

- List the pods in the `kube-public` namespace:

```
kubectl -n kube-public get pods
```

- Maybe it doesn't have pods, but what secrets is `kube-public` keeping?

Exercise

- List the secrets in the `kube-public` namespace:

```
kubectl -n kube-public get secrets
```

- `kube-public` is created by kubeadm & used for security bootstrapping

Setting up Kubernetes

Setting up Kubernetes

- How did we set up these Kubernetes clusters that we're using?

Setting up Kubernetes

- How did we set up these Kubernetes clusters that we're using?

- We used `kubeadm` on freshly installed VM instances running Ubuntu 16.04 LTS

  1. Install Docker

  2. Install Kubernetes packages

  3. Run `kubeadm init` on the first node (it deploys the control plane on that node)

  4. Set up Weave (the overlay network)
     (that step is just one `kubectl apply` command; discussed later)

  5. Run `kubeadm join` on the other nodes (with the token produced by `kubeadm init`)

  6. Copy the configuration file generated by `kubeadm init`

- Check the prepare VMs README for more details

`kubeadm` drawbacks

- Doesn't set up Docker or any other container engine
- Doesn't set up the overlay network
- Doesn't set up multi-master (no high availability)

`kubeadm` drawbacks

- Doesn't set up Docker or any other container engine

- Doesn't set up the overlay network

- Doesn't set up multi-master (no high availability)

  (At least ... not yet! Though it's experimental in 1.12.)

`kubeadm` drawbacks

- Doesn't set up Docker or any other container engine

- Doesn't set up the overlay network

- Doesn't set up multi-master (no high availability)

  (At least ... not yet! Though it's experimental in 1.12.)

- "It's still twice as many steps as setting up a Swarm cluster 😕" -- Jérôme

Other deployment options

- If you are on Azure: AKS

- If you are on Google Cloud: GKE

- If you are on AWS: EKS or kops

- On a local machine: minikube, kubespawn, Docker4Mac

- If you want something customizable: kubicorn

  Probably the closest to a multi-cloud/hybrid solution so far, but in development

Even more deployment options

- If you like Ansible: kubespray

- If you like Terraform: typhoon

- If you like Terraform and Puppet: tarmak

- You can also learn how to install every component manually, with the excellent tutorial Kubernetes The Hard Way

  Kubernetes The Hard Way is optimized for learning, which means taking the long route to ensure you understand each task required to bootstrap a Kubernetes cluster.

- There are also many commercial options available!

- For a longer list, check the Kubernetes documentation:
  it has a great guide to pick the right solution to set up Kubernetes.

Running our first containers on Kubernetes

Previous section | Back to table of contents | Next section

Running our first containers on Kubernetes

- First things first: we cannot run a container

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

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

Exercise

- Let's ping `1.1.1.1`, Cloudflare's public DNS resolver:

```
kubectl run pingpong --image alpine ping 1.1.1.1
```

Starting a simple pod with `kubectl run`

- We need to specify at least a *name* and the image we want to use

Exercise

- Let's ping `1.1.1.1`, Cloudflare's public DNS resolver:

```
kubectl run pingpong --image alpine ping 1.1.1.1
```

(Starting with Kubernetes 1.12, we get a message telling us that `kubectl run` is deprecated. Let's ignore it for now.)

Behind the scenes of `kubectl run`

- Let's look at the resources that were created by `kubectl run`

Exercise

- List most resource types:

```
kubectl get all
```

Behind the scenes of `kubectl run`

- Let's look at the resources that were created by `kubectl run`

Exercise

- List most resource types:

```
kubectl get all
```

We should see the following things:

- `deployment.apps/pingpong` (the *deployment* that we just created)
- `replicaset.apps/pingpong-xxxxxxxxxx` (a *replica set* created by the deployment)
- `pod/pingpong-xxxxxxxxxx-yyyyy` (a *pod* created by the replica set)

Note: as of 1.10.1, resource types are displayed in more detail.

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*, `deployment.apps/pingpong`

```
NAME                       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/pingpong   1         1         1            1           10m
```

- That deployment created a *replica set*, `replicaset.apps/pingpong-xxxxxxxxxx`

```
NAME                                  DESIRED   CURRENT   READY     AGE
replicaset.apps/pingpong-7c8bbcd9bc   1         1         1         10m
```

- That replica set created a *pod*, `pod/pingpong-xxxxxxxxxx-yyyyy`

```
NAME                             READY     STATUS    RESTARTS   AGE
pod/pingpong-7c8bbcd9bc-6c9qz    1/1       Running   0          10m
```

- 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!)

Exercise

- 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

Exercise

- 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 (I mean, our pod) with `kubectl scale`

Exercise

- Scale our `pingpong` deployment:

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

Note: what if we tried to scale `replicaset.apps/pingpong-xxxxxxxxxx`?

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?

Exercise

- In a separate window, list pods, and keep watching them:

```
kubectl get pods -w
```

- Destroy a pod:

```
kubectl delete pod pingpong-xxxxxxxxxx-yyyyy
```

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*

What about that deprecation warning?

- As we can see from the previous slide, `kubectl run` can do many things
- The exact type of resource created is not obvious
- To make things more explicit, it is better to use `kubectl create`:
  - `kubectl create deployment` to create a deployment
  - `kubectl create job` to create a job
- Eventually, `kubectl run` will be used only to start one-shot pods

  (see https://github.com/kubernetes/kubernetes/pull/68132)

Various ways of creating resources

- `kubectl run`
  - easy way to get started
  - versatile
- `kubectl create <resource>`
  - explicit, but lacks some features
  - can't create a CronJob
  - can't pass command-line arguments to deployments
- `kubectl create -f foo.yaml` or `kubectl apply -f foo.yaml`
  - all features are available
  - requires writing YAML

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

Exercise

- 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.

Aren't we flooding 1.1.1.1?

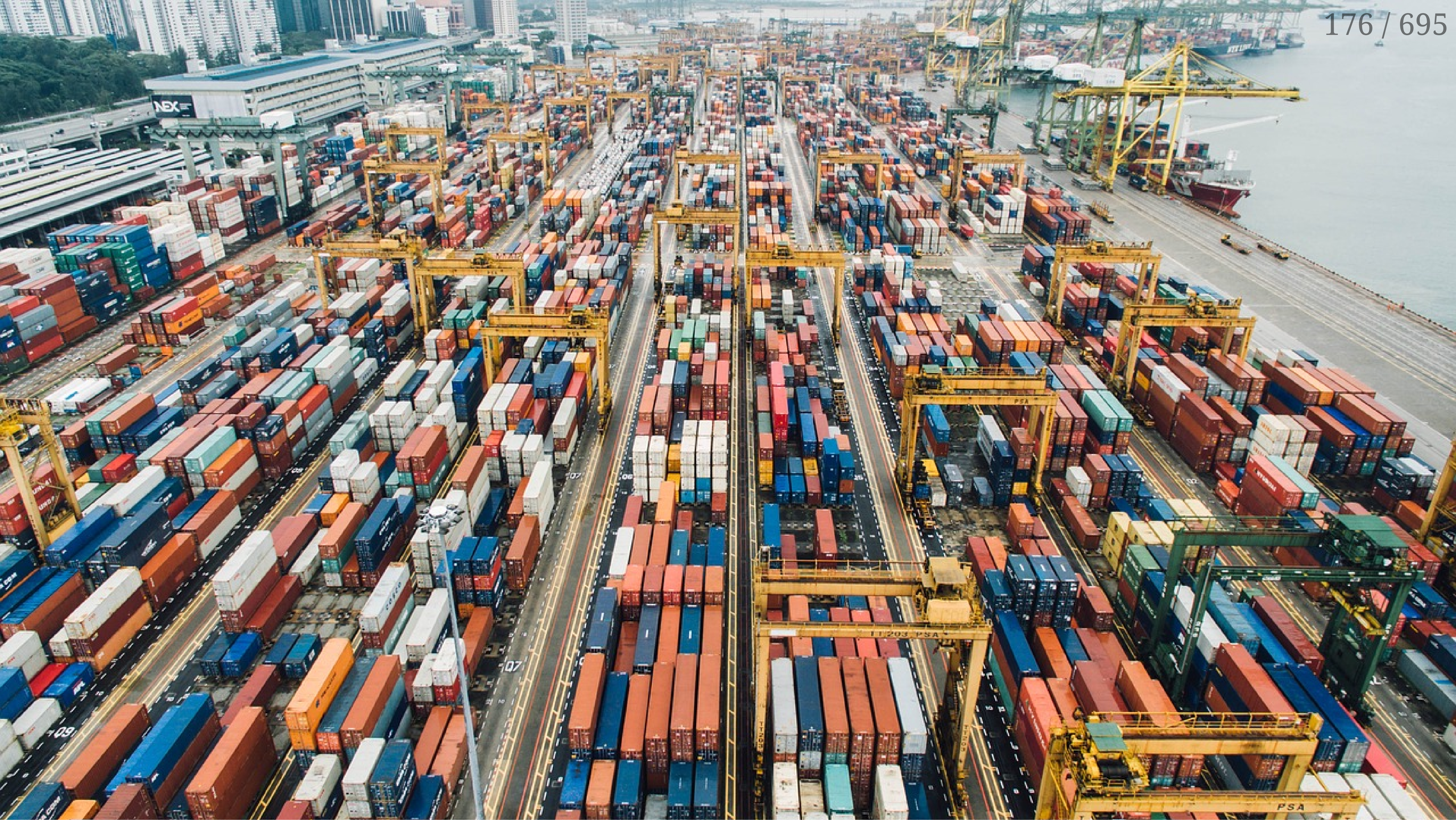
- If you're wondering this, good question!

- Don't worry, though:

  APNIC's research group held the IP addresses 1.1.1.1 and 1.0.0.1. While the addresses were valid, so many people had entered them into various random systems that they were continuously overwhelmed by a flood of garbage traffic. APNIC wanted to study this garbage traffic but any time they'd tried to announce the IPs, the flood would overwhelm any conventional network.

  (Source: https://blog.cloudflare.com/announcing-1111/)

- It's very unlikely that our concerted pings manage to produce even a modest blip at Cloudflare's NOC!



Exposing containers

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, CoreDNS 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)
    - available only when the underlying infrastructure provides some "load balancer as a service"
      (e.g. AWS, Azure, GCE, OpenStack...)

- `ExternalName`
    - the DNS entry managed by CoreDNS 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

Exercise

- Start a bunch of HTTP servers:

```
kubectl run httpenv --image=jpetazzo/httpenv --replicas=10
```

- Watch them being started:

```
kubectl get pods -w
```

The `jpetazzo/httpenv` image runs an HTTP server on port 8888.
It serves its environment variables in JSON format.

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

Exposing our deployment

- We'll create a default `ClusterIP` service

Exercise

- Expose the HTTP port of our server:

```
kubectl expose deploy/httpenv --port 8888
```

- 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 pods

Exercise

- Let's obtain the IP address that was allocated for our service, *programmatically:*

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

- Send a few requests:

```
curl http://$IP:8888/
```

- Too much output? Filter it with `jq`:

```
curl -s http://$IP:8888/ | jq .HOSTNAME
```

Testing our service

- We will now send a few HTTP requests to our pods

Exercise

- Let's obtain the IP address that was allocated for our service, *programmatically:*

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

- Send a few requests:

```
curl http://$IP:8888/
```

- Too much output? Filter it with `jq`:

```
curl -s http://$IP:8888/ | jq .HOSTNAME
```

Try it a few times! Our requests are load balanced across multiple pods.

If we don't need a load balancer

- Sometimes, we want to access our scaled services directly:
  - if we want to save a tiny little bit of latency (typically less than 1ms)
  - if we need to connect over arbitrary ports (instead of a few fixed ones)
  - if we need to communicate over another protocol than UDP or TCP
  - if we want to decide how to balance the requests client-side
  - ...
- In that case, we can use a "headless service"

Headless services

- A headless service is obtained by setting the `clusterIP` field to `None`

  (Either with `--cluster-ip=None`, or by providing a custom YAML)

- As a result, the service doesn't have a virtual IP address

- Since there is no virtual IP address, there is no load balancer either

- CoreDNS will return the pods' IP addresses as multiple `A` records

- This gives us an easy way to discover all the replicas for a deployment

Services and endpoints

- A service has a number of "endpoints"
- Each endpoint is a host + port where the service is available
- The endpoints are maintained and updated automatically by Kubernetes

Exercise

- Check the endpoints that Kubernetes has associated with our `httpenv` service:

```
kubectl describe service httpenv
```

In the output, there will be a line starting with `Endpoints:`.

That line will list a bunch of addresses in `host:port` format.

Viewing endpoint details

- When we have many endpoints, our display commands truncate the list

```
kubectl get endpoints
```

- If we want to see the full list, we can use one of the following commands:

```
kubectl describe endpoints httpenv
kubectl get endpoints httpenv -o yaml
```

- These commands will show us a list of IP addresses

- These IP addresses should match the addresses of the corresponding pods:

```
kubectl get pods -l run=httpenv -o wide
```

`endpoints` not `endpoint`

- `endpoints` is the only resource that cannot be singular

```
$ kubectl get endpoint
error: the server doesn't have a resource type "endpoint"
```

- This is because the type itself is plural (unlike every other resource)

- There is no `endpoint` object: `type Endpoints struct`

- The type doesn't represent a single endpoint, but a list of endpoints

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 `$REGISTRY/servicename`
- Upload them to a registry
- Create deployments using the images
- Expose (with a ClusterIP) the services that need to communicate
- Expose (with a NodePort) the WebUI

Which registry do we want to use?

- We could use the Docker Hub

- Or a service offered by our cloud provider (ACR, GCR, ECR...)

- Or we could just self-host that registry

We'll self-host the registry because it's the most generic solution for this workshop.

Using the open source registry

- We need to run a `registry` container

- 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 a NodePort,
  so that it's available through `127.0.0.1:xxxxx` on each node

Deploying a self-hosted registry

Deploying a self-hosted registry

- We will deploy a registry container, and expose it with a NodePort

Exercise

- Create the registry service:

```
kubectl run registry --image=registry
```

- Expose it on a NodePort:

```
kubectl expose deploy/registry --port=5000 --type=NodePort
```

Connecting to our registry

- We need to find out which port has been allocated

Exercise

- View the service details:

```
kubectl describe svc/registry
```

- Get the port number programmatically:

```
NODEPORT=$(kubectl get svc/registry -o json | jq .spec.ports[0].nodePort)
REGISTRY=127.0.0.1:$NODEPORT
```

Testing our registry

- A convenient Docker registry API route to remember is `/v2/_catalog`

Exercise

- View the repositories currently held in our registry:

```
curl $REGISTRY/v2/_catalog
```

Testing our registry

- A convenient Docker registry API route to remember is `/v2/_catalog`

⌨️ Exercise

- View the repositories currently held in our registry:

```
curl $REGISTRY/v2/_catalog
```

We should see:

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

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 $REGISTRY/busybox
```

- Push it:

```
docker push $REGISTRY/busybox
```

Checking again what's on our local registry

- Let's use the same endpoint as before

Exercise

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

```
curl $REGISTRY/v2/_catalog
```

The curl command should now output:

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

Building and pushing our images

- We are going to use a convenient feature of Docker Compose

Exercise

- Go to the `stacks` directory:

```
cd ~/container.training/stacks
```

- Build and push the images:

```
export REGISTRY
export TAG=v0.1
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.

```yaml
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
```

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

Avoiding the `latest` tag

⚠️ Make sure that you've set the `TAG` variable properly!

- If you don't, the tag will default to `latest`
- The problem with `latest`: nobody knows what it points to!
  - the latest commit in the repo?
  - the latest commit in some branch? (Which one?)
  - the latest tag?
  - some random version pushed by a random team member?
- If you keep pushing the `latest` tag, how do you roll back?
- Image tags should be meaningful, i.e. correspond to code branches, tags, or hashes

Deploying all the things

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

Exercise

- Deploy `redis`:

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

- Deploy everything else:

```
for SERVICE in hasher rng webui worker; do
  kubectl run $SERVICE --image=$REGISTRY/$SERVICE:$TAG
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)

Exercise

- Look at some logs:

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

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)

⌨️ Exercise

- Look at some logs:

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

🤔 `rng` is fine ... But not `worker`.

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)

⌨️ Exercise

- 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 internally

Previous section | Back to table of contents | Next section

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

Exercise

- 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

Exercise

- Stream the worker's logs:

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

(Give it about 10 seconds to recover)

Is this working yet?

- The `worker` has an infinite loop, that retries 10 seconds after an error

Exercise

- 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

Previous section | Back to table of contents | Next section

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)

Exercise

- Create a `NodePort` service for the Web UI:

  ```
  kubectl expose deploy/webui --type=NodePort --port=80
  ```

- 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

Exercise

- Open the web UI in your browser (http://node-ip-address:3xxxx/)

Accessing the web UI

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

Exercise

- Open the web UI in your browser (http://node-ip-address:3xxxx/)

Yes, this may take a little while to update. *(Narrator: it was DNS.)*

Accessing the web UI

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

Exercise

- Open the web UI in your browser (http://node-ip-address:3xxxx/)

Yes, this may take a little while to update. *(Narrator: it was DNS.)*

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


ECS
European Containers
ECS
European Containers.com

Accessing the API with `kubectl proxy`

Previous section | Back to table of contents | Next section

Accessing the API with `kubectl proxy`

- The API requires us to authenticate[1]

- There are many authentication methods available, including:

  - TLS client certificates
    (that's what we've used so far)

  - HTTP basic password authentication
    (from a static file; not recommended)

  - various token mechanisms
    (detailed in the documentation)

[1]OK, we lied. If you don't authenticate, you are considered to be user `system:anonymous`, which doesn't have any access rights by default.

Accessing the API directly

- Let's see what happens if we try to access the API directly with `curl`

Exercise

- Retrieve the ClusterIP allocated to the `kubernetes` service:

```
kubectl get svc kubernetes
```

- Replace the IP below and try to connect with `curl`:

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

The API will tell us that user `system:anonymous` cannot access this path.

Authenticating to the API

If we wanted to talk to the API, we would need to:

- extract our TLS key and certificate information from `~/.kube/config`

  (the information is in PEM format, encoded in base64)

- use that information to present our certificate when connecting

  (for instance, with `openssl s_client -key ... -cert ... -connect ...`)

- figure out exactly which credentials to use

  (once we start juggling multiple clusters)

- change that whole process if we're using another authentication method

🤔 There has to be a better way!

Using `kubectl proxy` for authentication

- `kubectl proxy` runs a proxy in the foreground

- This proxy lets us access the Kubernetes API without authentication

  (`kubectl proxy` adds our credentials on the fly to the requests)

- This proxy lets us access the Kubernetes API over plain HTTP

- This is a great tool to learn and experiment with the Kubernetes API

- ... And for serious usages as well (suitable for one-shot scripts)

- For unattended use, it is better to create a service account

Trying `kubectl proxy`

- Let's start `kubectl proxy` and then do a simple request with `curl`!

Exercise

- Start `kubectl proxy` in the background:

```
kubectl proxy &
```

- Access the API's default route:

```
curl localhost:8001
```

- Terminate the proxy:

```
kill %1
```

The output is a list of available API routes.

`kubectl proxy` is intended for local use

- By default, the proxy listens on port 8001

  (But this can be changed, or we can tell `kubectl proxy` to pick a port)

- By default, the proxy binds to `127.0.0.1`

  (Making it unreachable from other machines, for security reasons)

- By default, the proxy only accepts connections from:

  `^localhost$,^127\.0\.0\.1$,^\[::1\]$`

- This is great when running `kubectl proxy` locally

- Not-so-great when you want to connect to the proxy from a remote machine

Running `kubectl proxy` on a remote machine

- If we wanted to connect to the proxy from another machine, we would need to:
  - bind to `INADDR_ANY` instead of `127.0.0.1`
  - accept connections from any address
- This is achieved with:

```
kubectl proxy --port=8888 --address=0.0.0.0 --accept-hosts=.*
```

⚠️ Do not do this on a real cluster: it opens full unauthenticated access!

Security considerations

- Running `kubectl proxy` openly is a huge security risk

- It is slightly better to run the proxy where you need it

  (and copy credentials, e.g. `~/.kube/config`, to that place)

- It is even better to use a limited account with reduced permissions

Good to know ...

- `kubectl proxy` also gives access to all internal services

- Specifically, services are exposed as such:

  ```
  /api/v1/namespaces/<namespace>/services/<service>/proxy
  ```

- We can use `kubectl proxy` to access an internal service in a pinch

  (or, for non HTTP services, `kubectl port-forward`)

- This is not very useful when running `kubectl` directly on the cluster

  (since we could connect to the services directly anyway)

- But it is very powerful as soon as you run `kubectl` from a remote machine

Controlling the cluster remotely

Controlling the cluster remotely

- All the operations that we do with `kubectl` can be done remotely

- In this section, we are going to use `kubectl` from our local machine

Installing `kubectl`

- If you already have `kubectl` on your local machine, you can skip this

Exercise

- Download the `kubectl` binary from one of these links:

  Linux | macOS | Windows

- On Linux and macOS, make the binary executable with `chmod +x kubectl`

  (And remember to run it with `./kubectl` or move it to your `$PATH`)

Note: if you are following along with a different platform (e.g. Linux on an architecture different from amd64, or with a phone or tablet), installing `kubectl` might be more complicated (or even impossible) so feel free to skip this section.

Testing `kubectl`

- Check that `kubectl` works correctly

  (before even trying to connect to a remote cluster!)

Exercise

- Ask `kubectl` to show its version number:

```
kubectl version --client
```

The output should look like this:

```
Client Version: version.Info{Major:"1", Minor:"11", GitVersion:"v1.11.2",
GitCommit:"bb9ffb1654d4a729bb4cec18ff088eacc153c239", GitTreeState:"clean",
BuildDate:"2018-08-07T23:17:28Z", GoVersion:"go1.10.3", Compiler:"gc",
Platform:"linux/amd64"}
```

Moving away the existing `~/.kube/config`

- If you already have a `~/.kube/config` file, move it away

  (we are going to overwrite it in the following slides!)

- If you never used `kubectl` on your machine before: nothing to do!

- If you already used `kubectl` to control a Kubernetes cluster before:

  - rename `~/.kube/config` to e.g. `~/.kube/config.bak`

Copying the configuration file from `node1`

- The `~/.kube/config` file that is on `node1` contains all the credentials we need

- Let's copy it over!

Exercise

- Copy the file from `node1`; if you are using macOS or Linux, you can do:

```
scp USER@X.X.X.X:.kube/config ~/.kube/config
# Make sure to replace X.X.X.X with the IP address of node1,
# and USER with the user name used to log into node1!
```

- If you are using Windows, adapt these instructions to your SSH client

Updating the server address

- There is a good chance that we need to update the server address
- To know if it is necessary, run `kubectl config view`
- Look for the `server:` address:
  - if it matches the public IP address of `node1`, you're good!
  - if it is anything else (especially a private IP address), update it!
- To update the server address, run:

```
kubectl config set-cluster kubernetes --server=https://X.X.X.X:6443
kubectl config set-cluster kubernetes --insecure-skip-tls-verify
# Make sure to replace X.X.X.X with the IP address of node1!
```

Why do we skip TLS verification?

- Generally, the Kubernetes API uses a certificate that is valid for:
  - `kubernetes`
  - `kubernetes.default`
  - `kubernetes.default.svc`
  - `kubernetes.default.svc.cluster.local`
  - the ClusterIP address of the `kubernetes` service
  - the hostname of the node hosting the control plane (e.g. `node1`)
  - the IP address of the node hosting the control plane
- On most clouds, the IP address of the node is an internal IP address
- ... And we are going to connect over the external IP address
- ... And that external IP address was not used when creating the certificate!

⚠️ It's better to NOT skip TLS verification; this is for educational purposes only!

Checking that we can connect to the cluster

- We can now run a couple of trivial commands to check that all is well

Exercise

- Check the versions of the local client and remote server:

```
kubectl version
```

- View the nodes of the cluster:

```
kubectl get nodes
```

We can now utilize the cluster exactly as we did before, ignoring that it's remote.

Accessing internal services

Accessing internal services

- When we are logged in on a cluster node, we can access internal services

  (by virtue of the Kubernetes network model: all nodes can reach all pods and services)

- When we are accessing a remote cluster, things are different

  (generally, our local machine won't have access to the cluster's internal subnet)

- How can we temporarily access a service without exposing it to everyone?

Accessing internal services

- When we are logged in on a cluster node, we can access internal services

  (by virtue of the Kubernetes network model: all nodes can reach all pods and services)

- When we are accessing a remote cluster, things are different

  (generally, our local machine won't have access to the cluster's internal subnet)

- How can we temporarily access a service without exposing it to everyone?

- `kubectl proxy`: gives us access to the API, which includes a proxy for HTTP resources

- `kubectl port-forward`: allows forwarding of TCP ports to arbitrary pods, services, ...

Suspension of disbelief

The exercises in this section assume that we have set up `kubectl` on our local machine in order to access a remote cluster.

We will therefore show how to access services and pods of the remote cluster, from our local machine.

You can also run these exercises directly on the cluster (if you haven't installed and set up `kubectl` locally).

Running commands locally will be less useful (since you could access services and pods directly), but keep in mind that these commands will work anywhere as long as you have installed and set up `kubectl` to communicate with your cluster.

`kubectl proxy` in theory

- Running `kubectl proxy` gives us access to the entire Kubernetes API

- The API includes routes to proxy HTTP traffic

- These routes look like the following:

  `/api/v1/namespaces/<namespace>/services/<service>/proxy`

- We just add the URI to the end of the request, for instance:

  `/api/v1/namespaces/<namespace>/services/<service>/proxy/index.html`

- We can access `services` and `pods` this way

`kubectl proxy` in practice

- Let's access the `webui` service through `kubectl proxy`

Exercise

- Run an API proxy in the background:

```
kubectl proxy &
```

- Access the `webui` service:

```
curl localhost:8001/api/v1/namespaces/default/services/webui/proxy/index.html
```

- Terminate the proxy:

```
kill %1
```

`kubectl port-forward` in theory

- What if we want to access a TCP service?
- We can use `kubectl port-forward` instead
- It will create a TCP relay to forward connections to a specific port

  (of a pod, service, deployment...)
- The syntax is:

  `kubectl port-forward service/name_of_service local_port:remote_port`
- If only one port number is specified, it is used for both local and remote ports

`kubectl port-forward` in practice

- Let's access our remote Redis server

Exercise

- Forward connections from local port 10000 to remote port 6379:

```
kubectl port-forward svc/redis 10000:6379 &
```

- Connect to the Redis server:

```
telnet localhost 10000
```

- Issue a few commands, e.g. `INFO server` then `QUIT`

- Terminate the port forwarder:

```
kill %1
```


한진 보스톤
HANJIN BOSTON
MAJURO

The Kubernetes dashboard

The Kubernetes dashboard

- Kubernetes resources can also be viewed with a web dashboard

- We are going to deploy that dashboard with *three commands:*

  1) actually *run* the dashboard

  2) bypass SSL for the dashboard

  3) bypass authentication for the dashboard

The Kubernetes dashboard

- Kubernetes resources can also be viewed with a web dashboard

- We are going to deploy that dashboard with *three commands:*

  1) actually *run* the dashboard

  2) bypass SSL for the dashboard

  3) bypass authentication for the dashboard

There is an additional step to make the dashboard available from outside (we'll get to that)

The Kubernetes dashboard

- Kubernetes resources can also be viewed with a web dashboard
- We are going to deploy that dashboard with *three commands:*

  1) actually *run* the dashboard

  2) bypass SSL for the dashboard

  3) bypass authentication for the dashboard

There is an additional step to make the dashboard available from outside (we'll get to that)

⚠️ Yes, this will open our cluster to all kinds of shenanigans. Don't do this at home.

1) Running the dashboard

- We need to create a *deployment* and a *service* for the dashboard
- But also a *secret*, a *service account*, a *role* and a *role binding*
- All these things can be defined in a YAML file and created with `kubectl apply -f`

Exercise

- Create all the dashboard resources, with the following command:

```
kubectl apply -f ~/container.training/k8s/kubernetes-dashboard.yaml
```

2) Bypassing SSL for the dashboard

- The Kubernetes dashboard uses HTTPS, but we don't have a certificate
- Recent versions of Chrome (63 and later) and Edge will refuse to connect

  (You won't even get the option to ignore a security warning!)
- We could (and should!) get a certificate, e.g. with Let's Encrypt
- ... But for convenience, for this workshop, we'll forward HTTP to HTTPS

⚠️ Do not do this at home, or even worse, at work!

Running the SSL unwrapper

- We are going to run `socat`, telling it to accept TCP connections and relay them over SSL
- Then we will expose that `socat` instance with a `NodePort` service
- For convenience, these steps are neatly encapsulated into another YAML file

Exercise

- Apply the convenient YAML file, and defeat SSL protection:

```
kubectl apply -f ~/container.training/k8s/socat.yaml
```

All our dashboard traffic is now clear-text, including passwords!

Connecting to the dashboard

Exercise

- Check which port the dashboard is on:

```
kubectl -n kube-system get svc socat
```

You'll want the `3xxxx` port.

Exercise

- Connect to http://oneofournodes:3xxxx/

The dashboard will then ask you which authentication you want to use.

Dashboard authentication

- We have three authentication options at this point:
  - token (associated with a role that has appropriate permissions)
  - kubeconfig (e.g. using the `~/.kube/config` file from `node1`)
  - "skip" (use the dashboard "service account")
- Let's use "skip": we get a bunch of warnings and don't see much

3) Bypass authentication for the dashboard

- The dashboard documentation explains how to do this

- We just need to load another YAML file!

Exercise

- Grant admin privileges to the dashboard so we can see our resources:

```
kubectl apply -f ~/container.training/k8s/grant-admin-to-dashboard.yaml
```

- Reload the dashboard and enjoy!

3) Bypass authentication for the dashboard

- The dashboard documentation explains how to do this
- We just need to load another YAML file!

Exercise

- Grant admin privileges to the dashboard so we can see our resources:

```
kubectl apply -f ~/container.training/k8s/grant-admin-to-dashboard.yaml
```

- Reload the dashboard and enjoy!

⚠️ By the way, we just added a backdoor to our Kubernetes cluster!

Exposing the dashboard over HTTPS

- We took a shortcut by forwarding HTTP to HTTPS inside the cluster
- Let's expose the dashboard over HTTPS!
- The dashboard is exposed through a `ClusterIP` service (internal traffic only)
- We will change that into a `NodePort` service (accepting outside traffic)

Exercise

- Edit the service:

```
kubectl edit service kubernetes-dashboard
```

Exposing the dashboard over HTTPS

- We took a shortcut by forwarding HTTP to HTTPS inside the cluster

- Let's expose the dashboard over HTTPS!

- The dashboard is exposed through a `ClusterIP` service (internal traffic only)

- We will change that into a `NodePort` service (accepting outside traffic)

Exercise

- Edit the service:

```
kubectl edit service kubernetes-dashboard
```

`NotFound`?!? Y U NO WORK?!?

Editing the `kubernetes-dashboard` service

- If we look at the YAML that we loaded before, we'll get a hint

Editing the `kubernetes-dashboard` service

- If we look at the YAML that we loaded before, we'll get a hint

- The dashboard was created in the `kube-system` namespace

Editing the `kubernetes-dashboard` service

- If we look at the YAML that we loaded before, we'll get a hint
- The dashboard was created in the `kube-system` namespace

Exercise

- Edit the service:

```
kubectl -n kube-system edit service kubernetes-dashboard
```

- Change type `type:` from `ClusterIP` to `NodePort`, save, and exit
- Check the port that was assigned with `kubectl -n kube-system get services`
- Connect to https://oneofournodes:3xxxx/ (yes, https)

Running the Kubernetes dashboard securely

- The steps that we just showed you are *for educational purposes only!*

- If you do that on your production cluster, people can and will abuse it

- For an in-depth discussion about securing the dashboard,
  check this excellent post on Heptio's blog

Security implications of `kubectl apply`

Previous section | Back to table of contents | Next section

Security implications of `kubectl apply`

- When we do `kubectl apply -f <URL>`, we create arbitrary resources

- Resources can be evil; imagine a `deployment` that …

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

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

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

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)

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

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` 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

- Example: the official setup instructions for most pod networks

`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
- Example: the official setup instructions for most pod networks
- It introduces new failure modes (like if you try to apply yaml from a link that's no longer valid)


© Lode Van de Velde

Scaling a deployment

Scaling a deployment

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

Exercise

- Open two new terminals to check what's going on with pods and deployments:

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

- 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

Previous section | Back to table of contents | Next section

Daemon sets

- We want to scale `rng` in a way that is different from how we scaled `worker`
- We want one (and exactly one) instance of `rng` per node
- What if we just scale up `deploy/rng` to the number of nodes?
  - nothing guarantees that the `rng` containers will be distributed evenly
  - if we add nodes later, they will not automatically run a copy of `rng`
  - if we remove (or reboot) a node, one `rng` container will restart elsewhere
- Instead of a `deployment`, we will use a `daemonset`

Daemon sets in practice

- 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 1.12, the CLI cannot create daemon sets

Creating a daemon set

- Unfortunately, as of Kubernetes 1.12, the CLI cannot create daemon sets

- More precisely: it doesn't have a subcommand to create a daemon set

Creating a daemon set

- Unfortunately, as of Kubernetes 1.12, 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
```

Creating a daemon set

- Unfortunately, as of Kubernetes 1.12, 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?

Creating a daemon set

- Unfortunately, as of Kubernetes 1.12, 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

Creating a daemon set

- Unfortunately, as of Kubernetes 1.12, 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

Exercise

- 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?)

Exercise

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

- Save, quit

- Try to create our new resource:

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

"Casting" a resource to another

- What if we just changed the `kind` field?

  (It can't be that easy, right?)

Exercise

- 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,
...
```

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

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 `progressDeadlineSeconds` field (also used by the rollout mechanism)
  - remove the `status: {}` line at the end

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 `progressDeadlineSeconds` field (also used by the rollout mechanism)
  - 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's actual name is `--validate=false`

Exercise

- 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's actual name is `--validate=false`

⌨️ Exercise

- 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's actual name is `--validate=false`

⌨️ Exercise

- 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`?

Exercise

- Look at the resources that we have now:

```
kubectl get all
```

Checking what we've done

- Did we transform our `deployment` into a `daemonset`?

Exercise

- Look at the resources that we have now:

```
kubectl get all
```

We have two resources called `rng`:

- the *deployment* that was existing before

- the *daemon set* that we just created

We also have one too many pods.
(The pod corresponding to the *deployment* still exists.)

deploy/rng and ds/rng

- You can have different resource types with the same name

  (i.e. a *deployment* and a *daemon set* both named rng)

- We still have the old rng *deployment*

```
NAME                             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/rng              1         1         1            1           18m
```

- But now we have the new rng *daemon set* as well

```
NAME                   DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
daemonset.apps/rng     2         2         2       2            2           <none>          9s
```

Too many pods

- If we check with `kubectl get pods`, we see:
  - *one pod* for the deployment (named `rng-xxxxxxxxxx-yyyyy`)
  - *one pod per node* for the daemon set (named `rng-zzzzz`)

```
NAME                         READY     STATUS    RESTARTS   AGE
rng-54f57d4d49-7pt82         1/1       Running   0          11m
rng-b85tm                    1/1       Running   0          25s
rng-hfbrr                    1/1       Running   0          25s
[...]
```

Too many pods

- If we check with `kubectl get pods`, we see:
  - *one pod* for the deployment (named `rng-xxxxxxxxxx-yyyyy`)
  - *one pod per node* for the daemon set (named `rng-zzzzz`)

```
NAME                        READY     STATUS    RESTARTS   AGE
rng-54f57d4d49-7pt82        1/1       Running   0          11m
rng-b85tm                   1/1       Running   0          25s
rng-hfbrr                   1/1       Running   0          25s
[...]
```

The daemon set created one pod per node, except on the master node.

The master node has taints preventing pods from running there.

(To schedule a pod on this node anyway, the pod will require appropriate tolerations.)
(Off by one? We don't run these pods on the node hosting the control plane.)

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

Exercise

- Check the logs of all the pods having a label `run=rng`:

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

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

Exercise

- 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.

The magic of selectors

- The `rng` *service* is load balancing requests to a set of pods

- This set of pods is defined as "pods having the label `run=rng`"

Exercise

- Check the *selector* in the `rng` service definition:

```
kubectl describe service rng
```

When we created additional pods with this label, they were automatically detected by `svc/rng` and added as *endpoints* to the associated load balancer.

Removing the first pod from the load balancer

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

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.

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?

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.

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*.

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?

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*

Exercise

- Show detailed information about the `rng` deployment:

```
kubectl describe deploy rng
```

- Show detailed information about the `rng` replica:
  (The second command doesn't require you to get the exact name of the replica set)

```
kubectl describe rs rng-yyyyyyyy
kubectl describe rs -l run=rng
```

Deep dive into selectors

- Let's look at the selectors for the `rng` *deployment* and the associated *replica set*

Exercise

- Show detailed information about the `rng` deployment:

```
kubectl describe deploy rng
```

- Show detailed information about the `rng` replica:
  (The second command doesn't require you to get the exact name of the replica set)

```
kubectl describe rs rng-yyyyyyyy
kubectl describe rs -l run=rng
```

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

Updating a service through labels and selectors

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*

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`

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"`

Exercise

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

```
kubectl edit daemonset rng
```

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

```
kubectl edit service rng
```

Checking what we've done

Exercise

- Check the most recent log line of all `run=rng` pods to confirm that exactly one per node is now active:

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

The timestamps should give us a hint about how many pods are currently receiving traffic.

Exercise

- Look at the pods that we have right now:

```
kubectl get pods
```

Cleaning up

- The pods of the deployment and the "old" daemon set are still running

- We are going to identify them programmatically

Exercise

- List the pods with `run=rng` but without `isactive=yes`:

```
kubectl get pods -l run=rng,isactive!=yes
```

- Remove these pods:

```
kubectl delete pods -l run=rng,isactive!=yes
```

Cleaning up stale pods

```
$ kubectl get pods
NAME                        READY     STATUS        RESTARTS    AGE
rng-54f57d4d49-7pt82        1/1       Terminating   0           51m
rng-54f57d4d49-vgz9h        1/1       Running       0           22s
rng-b85tm                   1/1       Terminating   0           39m
rng-hfbrr                   1/1       Terminating   0           39m
rng-vplmj                   1/1       Running       0           7m
rng-xbpvg                   1/1       Running       0           7m
[...]
```

- The extra pods (noted `Terminating` above) are going away
- ... But a new one (`rng-54f57d4d49-vgz9h` above) was restarted immediately!

Cleaning up stale pods

```
$ kubectl get pods
NAME                          READY     STATUS        RESTARTS    AGE
rng-54f57d4d49-7pt82          1/1       Terminating   0           51m
rng-54f57d4d49-vgz9h          1/1       Running       0           22s
rng-b85tm                     1/1       Terminating   0           39m
rng-hfbrr                     1/1       Terminating   0           39m
rng-vplmj                     1/1       Running       0           7m
rng-xbpvg                     1/1       Running       0           7m
[...]
```

- The extra pods (noted `Terminating` above) are going away

- ... But a new one (`rng-54f57d4d49-vgz9h` above) was restarted immediately!

- Remember, the *deployment* still exists, and makes sure that one pod is up and running

- If we delete the pod associated to the deployment, it is recreated automatically

Deleting a deployment

Exercise

- Remove the `rng` deployment:

```
kubectl delete deployment rng
```

Deleting a deployment

⌨️ Exercise

- Remove the `rng` deployment:

```
kubectl delete deployment rng
```

- The pod that was created by the deployment is now being terminated:

```
$ kubectl get pods
NAME                     READY     STATUS        RESTARTS   AGE
rng-54f57d4d49-vgz9h     1/1       Terminating   0          4m
rng-vplmj                1/1       Running       0          11m
rng-xbpvg                1/1       Running       0          11m
[...]
```

Ding, dong, the deployment is dead! And the daemon set lives on.

Avoiding extra pods

- When we changed the definition of the daemon set, it immediately created new pods. We had to remove the old ones manually.

- How could we have avoided this?

Avoiding extra pods

- When we changed the definition of the daemon set, it immediately created new pods. We had to remove the old ones manually.

- How could we have avoided this?

- By adding the `isactive: "yes"` label to the pods before changing the daemon set!

- This can be done programmatically with `kubectl patch`:

```
PATCH='
metadata:
  labels:
    isactive: "yes"
'
kubectl get pods -l run=rng -l controller-revision-hash -o name |
  xargs kubectl patch -p "$PATCH"
```

Labels and debugging

- When a pod is misbehaving, we can delete it: another one will be recreated
- But we can also change its labels
- It will be removed from the load balancer (it won't receive traffic anymore)
- Another pod will be recreated immediately
- But the problematic pod is still here, and we can inspect and debug it
- We can even re-add it to the rotation if necessary

  (Very useful to troubleshoot intermittent and elusive bugs)

Labels and advanced rollout control

- Conversely, we can add pods matching a service's selector
- These pods will then receive requests and serve traffic
- Examples:
  - one-shot pod with all debug flags enabled, to collect logs
  - pods created automatically, but added to rotation in a second step (by setting their label accordingly)
- This gives us building blocks for canary and blue/green deployments

Rolling updates

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)_/¯

Rolling updates

- 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)

Checking current rollout parameters

- Recall how we build custom reports with `kubectl` and `jq`:

Exercise

- Show the rollout plan for our deployments:

```
kubectl get deploy -o json |
        jq ".items[] | {name:.metadata.name} + .spec.strategy.rollingUpdate"
```

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

Exercise

- Go to the `stack` directory:

```
cd ~/container.training/stacks
```

- Edit `dockercoins/worker/worker.py`; update the first `sleep` line to sleep 1 second

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

```
#export REGISTRY=localhost:3xxxx
export TAG=v0.2
docker-compose -f dockercoins.yml build
docker-compose -f dockercoins.yml push
```

Rolling out the new `worker` service

Exercise

- 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=$REGISTRY/worker:$TAG
```

Rolling out the new `worker` service

Exercise

- 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=$REGISTRY/worker:$TAG
```

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

Give it some time

- At first, it looks like nothing is happening (the graph remains at the same level)
- According to `kubectl get deploy -w`, the `deployment` was updated really quickly
- But `kubectl get pods -w` tells a different story
- The old `pods` are still here, and they stay in `Terminating` state for a while
- Eventually, they are terminated; and then the graph decreases significantly
- This delay is due to the fact that our worker doesn't handle signals
- Kubernetes sends a "polite" shutdown request to the worker, which ignores it
- After a grace period, Kubernetes gets impatient and kills the container

  (The grace period is 30 seconds, but can be changed if needed)

Rolling out something invalid

- What happens if we make a mistake?

Exercise

- 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
```

Rolling out something invalid

- What happens if we make a mistake?

Exercise

- 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.

(After a minute, it will stabilize to be 20-25% slower.)

What's going on with our rollout?

- Why is our app a bit slower?

- Because `MaxUnavailable=25%`

  ... So the rollout terminated 2 replicas out of 10 available

- Okay, but why do we see 5 new replicas being rolled out?

- Because `MaxSurge=25%`

  ... So in addition to replacing 2 replicas, the rollout is also starting 3 more

- It rounded down the number of MaxUnavailable pods conservatively,
  but the total number of pods being rolled out is allowed to be 25+25=50%

The nitty-gritty details

- We start with 10 pods running for the `worker` deployment

- Current settings: MaxUnavailable=25% and MaxSurge=25%

- When we start the rollout:

  - two replicas are taken down (as per MaxUnavailable=25%)
  - two others are created (with the new version) to replace them
  - three others are created (with the new version) per MaxSurge=25%)

- Now we have 8 replicas up and running, and 5 being deployed

- Our rollout is stuck at this point!

Checking the dashboard during the bad rollout

If you haven't deployed the Kubernetes dashboard earlier, just skip this slide.

Exercise

- Check which port the dashboard is on:

```
kubectl -n kube-system get svc socat
```

Note the `3xxxx` port.

Exercise

- Connect to http://oneofournodes:3xxxx/

Checking the dashboard during the bad rollout

If you haven't deployed the Kubernetes dashboard earlier, just skip this slide.

Exercise

- Check which port the dashboard is on:

```
kubectl -n kube-system get svc socat
```

Note the `3xxxx` port.

Exercise

- Connect to http://oneofournodes:3xxxx/

- We have failures in Deployments, Pods, and Replica Sets

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

Exercise

- 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`
  - be conservative on availability (always have desired number of available workers)
  - go slow on rollout speed (update only 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: $REGISTRY/worker:v0.1
  strategy:
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  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

Exercise

- Apply all our changes and wait for them to take effect:

```
kubectl patch deployment worker -p "
spec:
  template:
    spec:
      containers:
      - name: worker
        image: $REGISTRY/worker:v0.1
  strategy:
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  minReadySeconds: 10
"
kubectl rollout status deployment worker
kubectl get deploy -o json worker |
      jq "{name:.metadata.name} + .spec.strategy.rollingUpdate"
```

Healthchecks

Healthchecks

- Kubernetes provides two kinds of healthchecks: liveness and readiness
- Healthchecks are *probes* that apply to *containers* (not to pods)
- Each container can have two (optional) probes:
  - liveness = is this container dead or alive?
  - readiness = is this container ready to serve traffic?
- Different probes are available (HTTP, TCP, program execution)
- Let's see the difference and how to use them!

Liveness probe

- Indicates if the container is dead or alive
- A dead container cannot come back to life
- If the liveness probe fails, the container is killed

  (to make really sure that it's really dead; no zombies or undeads!)
- What happens next depends on the pod's `restartPolicy`:
  - `Never`: the container is not restarted
  - `OnFailure` or `Always`: the container is restarted

When to use a liveness probe

- To indicate failures that can't be recovered
  - deadlocks (causing all requests to time out)
  - internal corruption (causing all requests to error)
- If the liveness probe fails *N* consecutive times, the container is killed
- *N* is the `failureThreshold` (3 by default)

Readiness probe

- Indicates if the container is ready to serve traffic
- If a container becomes "unready" (let's say busy!) it might be ready again soon
- If the readiness probe fails:
  - the container is *not* killed
  - if the pod is a member of a service, it is temporarily removed
  - it is re-added as soon as the readiness probe passes again

When to use a readiness probe

- To indicate temporary failures
  - the application can only service *N* parallel connections
  - the runtime is busy doing garbage collection or initial data load
- The container is marked as "not ready" after `failureThreshold` failed attempts

  (3 by default)
- It is marked again as "ready" after `successThreshold` successful attempts

  (1 by default)

Different types of probes

- HTTP request
  - specify URL of the request (and optional headers)
  - any status code between 200 and 399 indicates success
- TCP connection
  - the probe succeeds if the TCP port is open
- arbitrary exec
  - a command is executed in the container
  - exit status of zero indicates success

Benefits of using probes

- Rolling updates proceed when containers are *actually ready*

  (as opposed to merely started)

- Containers in a broken state gets killed and restarted

  (instead of serving errors or timeouts)

- Overloaded backends get removed from load balancer rotation

  (thus improving response times across the board)

Example: HTTP probe

Here is a pod template for the `rng` web service of the DockerCoins app:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: rng-with-liveness
spec:
  containers:
  - name: rng
    image: dockercoins/rng:v0.1
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 10
      periodSeconds: 1
```

If the backend serves an error, or takes longer than 1s, 3 times in a row, it gets killed.

Example: exec probe

Here is a pod template for a Redis server:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-with-liveness
spec:
  containers:
  - name: redis
    image: redis
    livenessProbe:
      exec:
        command: ["redis-cli", "ping"]
```

If the Redis process becomes unresponsive, it will be killed.

Details about liveness and readiness probes

- Probes are executed at intervals of `periodSeconds` (default: 10)
- The timeout for a probe is set with `timeoutSeconds` (default: 1)
- A probe is considered successful after `successThreshold` successes (default: 1)
- A probe is considered failing after `failureThreshold` failures (default: 3)
- If a probe is not defined, it's as if there was an "always successful" probe

Accessing logs from the CLI

Accessing logs from the CLI

- The `kubectl logs` commands has limitations:
  - it cannot stream logs from multiple pods at a time
  - when showing logs from multiple pods, it mixes them all together
- We are going to see how to do it better

Doing it manually

- We *could* (if we were so inclined), write a program or script that would:
  - take a selector as an argument
  - enumerate all pods matching that selector (with `kubectl get -l ...`)
  - fork one `kubectl logs --follow ...` command per container
  - annotate the logs (the output of each `kubectl logs ...` process) with their origin
  - preserve ordering by using `kubectl logs --timestamps ...` and merge the output

Doing it manually

- We *could* (if we were so inclined), write a program or script that would:
  - take a selector as an argument
  - enumerate all pods matching that selector (with `kubectl get -l ...`)
  - fork one `kubectl logs --follow ...` command per container
  - annotate the logs (the output of each `kubectl logs ...` process) with their origin
  - preserve ordering by using `kubectl logs --timestamps ...` and merge the output
- We *could* do it, but thankfully, others did it for us already!

Stern

Stern is an open source project by Wercker.

From the README:

Stern allows you to tail multiple pods on Kubernetes and multiple containers within the pod. Each result is color coded for quicker debugging.

The query is a regular expression so the pod name can easily be filtered and you don't need to specify the exact id (for instance omitting the deployment id). If a pod is deleted it gets removed from tail and if a new pod is added it automatically gets tailed.

Exactly what we need!

Installing Stern

- Run `stern` (without arguments) to check if it's installed:

```
$ stern
Tail multiple pods and containers from Kubernetes

Usage:
stern pod-query [flags]
```

- If it is not installed, the easiest method is to download a binary release

- The following commands will install Stern on a Linux Intel 64 bit machine:

```
sudo curl -L -o /usr/local/bin/stern \
   https://github.com/wercker/stern/releases/download/1.8.0/stern_linux_amd64
sudo chmod +x /usr/local/bin/stern
```

Using Stern

- There are two ways to specify the pods for which we want to see the logs:
  - `-l` followed by a selector expression (like with many `kubectl` commands)
  - with a "pod query", i.e. a regex used to match pod names
- These two ways can be combined if necessary

Exercise

- View the logs for all the rng containers:

```
stern rng
```

Stern convenient options

- The `--tail N` flag shows the last `N` lines for each container

  (Instead of showing the logs since the creation of the container)

- The `-t` / `--timestamps` flag shows timestamps

- The `--all-namespaces` flag is self-explanatory

Exercise

- View what's up with the `weave` system containers:

```
stern --tail 1 --timestamps --all-namespaces weave
```

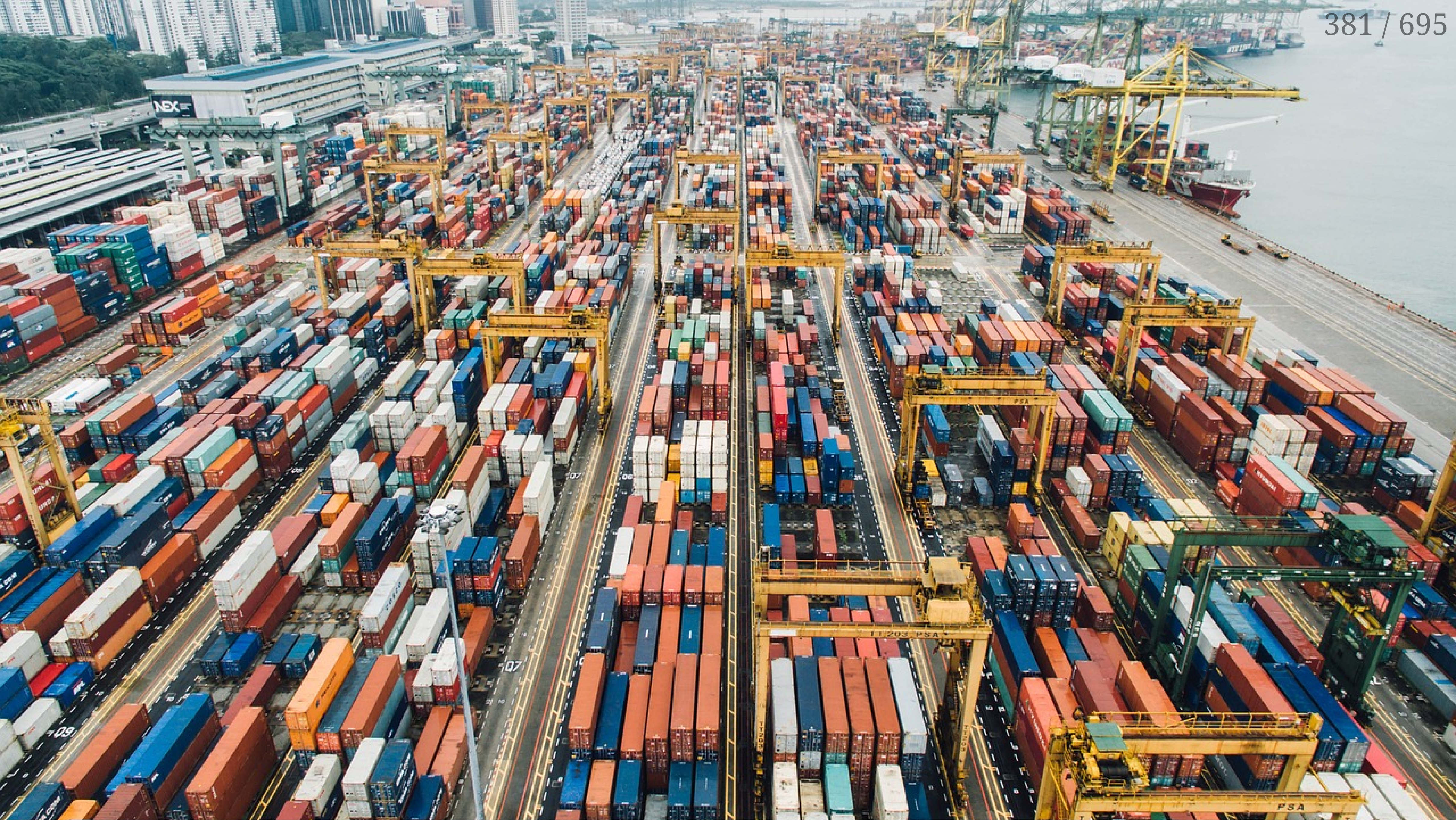
Using Stern with a selector

- When specifying a selector, we can omit the value for a label
- This will match all objects having that label (regardless of the value)
- Everything created with `kubectl run` has a label `run`
- We can use that property to view the logs of all the pods created with `kubectl run`

Exercise

- View the logs for all the things started with `kubectl run`:

```
stern -l run
```



Centralized logging

Centralized logging

- Using `kubectl` or `stern` is simple; but it has drawbacks:
  - when a node goes down, its logs are not available anymore
  - we can only dump or stream logs; we want to search/index/count...
- We want to send all our logs to a single place
- We want to parse them (e.g. for HTTP logs) and index them
- We want a nice web dashboard

Centralized logging

- Using `kubectl` or `stern` is simple; but it has drawbacks:
  - when a node goes down, its logs are not available anymore
  - we can only dump or stream logs; we want to search/index/count...
- We want to send all our logs to a single place
- We want to parse them (e.g. for HTTP logs) and index them
- We want a nice web dashboard
- We are going to deploy an EFK stack

What is EFK?

- EFK is three components:

  - ElasticSearch (to store and index log entries)

  - Fluentd (to get container logs, process them, and put them in ElasticSearch)

  - Kibana (to view/search log entries with a nice UI)

- The only component that we need to access from outside the cluster will be Kibana

Deploying EFK on our cluster

- We are going to use a YAML file describing all the required resources

Exercise

- Load the YAML file into our cluster:

```
kubectl apply -f ~/container.training/k8s/efk.yaml
```

If we look at the YAML file, we see that it creates a daemon set, two deployments, two services, and a few roles and role bindings (to give fluentd the required permissions).

The itinerary of a log line (before Fluentd)

- A container writes a line on stdout or stderr
- Both are typically piped to the container engine (Docker or otherwise)
- The container engine reads the line, and sends it to a logging driver
- The timestamp and stream (stdout or stderr) is added to the log line
- With the default configuration for Kubernetes, the line is written to a JSON file

  (`/var/log/containers/pod-name_namespace_container-id.log`)
- That file is read when we invoke `kubectl logs`; we can access it directly too

The itinerary of a log line (with Fluentd)

- Fluentd runs on each node (thanks to a daemon set)

- It binds-mounts `/var/log/containers` from the host (to access these files)

- It continuously scans this directory for new files; reads them; parses them

- Each log line becomes a JSON object, fully annotated with extra information: container id, pod name, Kubernetes labels ...

- These JSON objects are stored in ElasticSearch

- ElasticSearch indexes the JSON objects

- We can access the logs through Kibana (and perform searches, counts, etc.)

Accessing Kibana

- Kibana offers a web interface that is relatively straightforward

- Let's check it out!

⌨️ Exercise

- Check which `NodePort` was allocated to Kibana:

```
kubectl get svc kibana
```

- With our web browser, connect to Kibana

Using Kibana

Note: this is not a Kibana workshop! So this section is deliberately very terse.

- The first time you connect to Kibana, you must "configure an index pattern"
- Just use the one that is suggested, `@timestamp`*
- Then click "Discover" (in the top-left corner)
- You should see container logs
- Advice: in the left column, select a few fields to display, e.g.:

  `kubernetes.host`, `kubernetes.pod_name`, `stream`, `log`

*If you don't see `@timestamp`, it's probably because no logs exist yet.
Wait a bit, and double-check the logging pipeline!

Caveat emptor

We are using EFK because it is relatively straightforward to deploy on Kubernetes, without having to redeploy or reconfigure our cluster. But it doesn't mean that it will always be the best option for your use-case. If you are running Kubernetes in the cloud, you might consider using the cloud provider's logging infrastructure (if it can be integrated with Kubernetes).

The deployment method that we will use here has been simplified: there is only one ElasticSearch node. In a real deployment, you might use a cluster, both for performance and reliability reasons. But this is outside of the scope of this chapter.

The YAML file that we used creates all the resources in the `default` namespace, for simplicity. In a real scenario, you will create the resources in the `kube-system` namespace or in a dedicated namespace.

Managing stacks with Helm

Previous section | Back to table of contents | Next section

Managing stacks with Helm

- We created our first resources with `kubectl run`, `kubectl expose` ...

- We have also created resources by loading YAML files with `kubectl apply -f`

- For larger stacks, managing thousands of lines of YAML is unreasonable

- These YAML bundles need to be customized with variable parameters

  (E.g.: number of replicas, image version to use ...)

- It would be nice to have an organized, versioned collection of bundles

- It would be nice to be able to upgrade/rollback these bundles carefully

- Helm is an open source project offering all these things!

Helm concepts

- `helm` is a CLI tool

- `tiller` is its companion server-side component

- A "chart" is an archive containing templatized YAML bundles

- Charts are versioned

- Charts can be stored on private or public repositories

Installing Helm

- If the `helm` CLI is not installed in your environment, install it

Exercise

- Check if `helm` is installed:

```
helm
```

- If it's not installed, run the following command:

```
curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash
```

Installing Tiller

- Tiller is composed of a *service* and a *deployment* in the `kube-system` namespace

- They can be managed (installed, upgraded…) with the `helm` CLI

Exercise

- Deploy Tiller:

```
helm init
```

If Tiller was already installed, don't worry: this won't break it.

At the end of the install process, you will see:

```
Happy Helming!
```

Fix account permissions

- Helm permission model requires us to tweak permissions

- In a more realistic deployment, you might create per-user or per-team service accounts, roles, and role bindings

Exercise

- Grant `cluster-admin` role to `kube-system:default` service account:

```
kubectl create clusterrolebinding add-on-cluster-admin \
  --clusterrole=cluster-admin --serviceaccount=kube-system:default
```

(Defining the exact roles and permissions on your cluster requires a deeper knowledge of Kubernetes' RBAC model. The command above is fine for personal and development clusters.)

View available charts

- A public repo is pre-configured when installing Helm

- We can view available charts with `helm search` (and an optional keyword)

Exercise

- View all available charts:

```
helm search
```

- View charts related to `prometheus`:

```
helm search prometheus
```

Install a chart

- Most charts use `LoadBalancer` service types by default

- Most charts require persistent volumes to store data

- We need to relax these requirements a bit

Exercise

- Install the Prometheus metrics collector on our cluster:

```
helm install stable/prometheus \
     --set server.service.type=NodePort \
     --set server.persistentVolume.enabled=false
```

Where do these `--set` options come from?

Inspecting a chart

- `helm inspect` shows details about a chart (including available options)

Exercise

- See the metadata and all available options for `stable/prometheus`:

```
helm inspect stable/prometheus
```

The chart's metadata includes an URL to the project's home page.

(Sometimes it conveniently points to the documentation for the chart.)

Creating a chart

- We are going to show a way to create a *very simplified* chart
- In a real chart, *lots of things* would be templatized

  (Resource names, service types, number of replicas...)

Exercise

- Create a sample chart:

  ```
  helm create dockercoins
  ```

- Move away the sample templates and create an empty template directory:

  ```
  mv dockercoins/templates dockercoins/default-templates
  mkdir dockercoins/templates
  ```

Exporting the YAML for our application

- The following section assumes that DockerCoins is currently running

Exercise

- Create one YAML file for each resource that we need:

```
while read kind name; do
  kubectl get -o yaml --export $kind $name > dockercoins/templates/$name-$kind.yaml
done <<EOF
deployment worker
deployment hasher
daemonset rng
deployment webui
deployment redis
service hasher
service rng
service webui
service redis
EOF
```

Testing our helm chart

Exercise

- Let's install our helm chart! (`dockercoins` is the path to the chart)

```
helm install dockercoins
```

Testing our helm chart

Exercise

- Let's install our helm chart! (`dockercoins` is the path to the chart)

```
helm install dockercoins
```

- Since the application is already deployed, this will fail:
  `Error: release loitering-otter failed: services "hasher" already exists`

- To avoid naming conflicts, we will deploy the application in another *namespace*

Namespaces

Previous section | Back to table of contents | Next section

Namespaces

- We cannot have two resources with the same name

  (Or can we...?)

Namespaces

- We cannot have two resources with the same name

  (Or can we...?)

- We cannot have two resources *of the same type* with the same name

  (But it's OK to have a `rng` service, a `rng` deployment, and a `rng` daemon set!)

Namespaces

- We cannot have two resources with the same name

  (Or can we...?)

- We cannot have two resources *of the same type* with the same name

  (But it's OK to have a `rng` service, a `rng` deployment, and a `rng` daemon set!)

- We cannot have two resources of the same type with the same name *in the same namespace*

  (But it's OK to have e.g. two `rng` services in different namespaces!)

Namespaces

- We cannot have two resources with the same name

  (Or can we...?)

- We cannot have two resources *of the same type* with the same name

  (But it's OK to have a `rng` service, a `rng` deployment, and a `rng` daemon set!)

- We cannot have two resources of the same type with the same name *in the same namespace*

  (But it's OK to have e.g. two `rng` services in different namespaces!)

- In other words: **the tuple *(type, name, namespace)* needs to be unique**

  (In the resource YAML, the type is called `Kind`)

Pre-existing namespaces

- If we deploy a cluster with `kubeadm`, we have three namespaces:
  - `default` (for our applications)
  - `kube-system` (for the control plane)
  - `kube-public` (contains one secret used for cluster discovery)
- If we deploy differently, we may have different namespaces

Creating namespaces

- Creating a namespace is done with the `kubectl create namespace` command:

```
kubectl create namespace blue
```

- We can also get fancy and use a very minimal YAML snippet, e.g.:

```
kubectl apply -f- <<EOF
apiVersion: v1
kind: Namespace
metadata:
  name: blue
EOF
```

- The two methods above are identical

- If we are using a tool like Helm, it will create namespaces automatically

Using namespaces

- We can pass a `-n` or `--namespace` flag to most `kubectl` commands:

```
kubectl -n blue get svc
```

- We can also use *contexts*
- A context is a *(user, cluster, namespace)* tuple
- We can manipulate contexts with the `kubectl config` command

Creating a context

- We are going to create a context for the `blue` namespace

Exercise

- View existing contexts to see the cluster name and the current user:

```
kubectl config get-contexts
```

- Create a new context:

```
kubectl config set-context blue --namespace=blue \
  --cluster=kubernetes --user=kubernetes-admin
```

We have created a context; but this is just some configuration values.

The namespace doesn't exist yet.

Using a context

- Let's switch to our new context and deploy the DockerCoins chart

Exercise

- Use the `blue` context:

```
kubectl config use-context blue
```

- Deploy DockerCoins:

```
helm install dockercoins
```

In the last command line, `dockercoins` is just the local path where we created our Helm chart before.

Viewing the deployed app

- Let's see if our Helm chart worked correctly!

Exercise

- Retrieve the port number allocated to the `webui` service:

```
kubectl get svc webui
```

- Point our browser to http://X.X.X.X:3xxxx

Note: it might take a minute or two for the app to be up and running.

Namespaces and isolation

- Namespaces *do not* provide isolation
- A pod in the `green` namespace can communicate with a pod in the `blue` namespace
- A pod in the `default` namespace can communicate with a pod in the `kube-system` namespace
- CoreDNS uses a different subdomain for each namespace
- Example: from any pod in the cluster, you can connect to the Kubernetes API with:

  `https://kubernetes.default.svc.cluster.local:443/`

Isolating pods

- Actual isolation is implemented with *network policies*
- Network policies are resources (like deployments, services, namespaces...)
- Network policies specify which flows are allowed:
  - between pods
  - from pods to the outside world
  - and vice-versa

Switch back to the default namespace

- Let's make sure that we don't run future exercises in the `blue` namespace

Exercise

- View the names of the contexts:

```
kubectl config get-contexts
```

- Switch back to the original context:

```
kubectl config use-context kubernetes-admin@kubernetes
```

Switching namespaces more easily

- Defining a new context for each namespace can be cumbersome

- We can also alter the current context with this one-liner:

```
kubectl config set-context --current --namespace=foo
```

- We can also use a little helper tool called `kubens`:

```
# Switch to namespace foo
kubens foo
# Switch back to the previous namespace
kubens -
```

`kubens` and `kubectx`

- With `kubens`, we can switch quickly between namespaces
- With `kubectx`, we can switch quickly between contexts
- Both tools are simple shell scripts available from https://github.com/ahmetb/kubectx
- On our clusters, they are installed as `kns` and `kctx`

  (for brevity and to avoid completion clashes between `kubectx` and `kubectl`)

`kube-ps1`

- It's easy to lose track of our current cluster / context / namespace

- `kube-ps1` makes it easy to track these, by showing them in our shell prompt

- It's a simple shell script availble from https://github.com/jonmosco/kube-ps1

- On our clusters, `kube-ps1` is installed and included in `PS1`:

  ```
  [123.45.67.89] (kubernetes-admin@kubernetes:default) docker@node1 ~
  ```

  (The highlighted part is `context:namespace`, managed by `kube-ps1`)

- Highly recommended if you work across multiple contexts or namespaces!

Network policies

Network policies

- Namespaces help us to *organize* resources
- Namespaces do not provide isolation
- By default, every pod can contact every other pod
- By default, every service accepts traffic from anyone
- If we want this to be different, we need *network policies*

What's a network policy?

A network policy is defined by the following things.

- A *pod selector* indicating which pods it applies to

  e.g.: "all pods in namespace `blue` with the label `zone=internal`"

- A list of *ingress rules* indicating which inbound traffic is allowed

  e.g.: "TCP connections to ports 8000 and 8080 coming from pods with label `zone=dmz`, and from the external subnet 4.42.6.0/24, except 4.42.6.5"

- A list of *egress rules* indicating which outbound traffic is allowed

A network policy can provide ingress rules, egress rules, or both.

How do network policies apply?

- A pod can be "selected" by any number of network policies

- If a pod isn't selected by any network policy, then its traffic is unrestricted

  (In other words: in the absence of network policies, all traffic is allowed)

- If a pod is selected by at least one network policy, then all traffic is blocked …

  … unless it is explicitly allowed by one of these network policies

Traffic filtering is flow-oriented

- Network policies deal with *connections*, not individual packets
- Example: to allow HTTP (80/tcp) connections to pod A, you only need an ingress rule

  (You do not need a matching egress rule to allow response traffic to go through)
- This also applies for UDP traffic

  (Allowing DNS traffic can be done with a single rule)
- Network policy implementations use stateful connection tracking

Pod-to-pod traffic

- Connections from pod A to pod B have to be allowed by both pods:
  - pod A has to be unrestricted, or allow the connection as an *egress* rule
  - pod B has to be unrestricted, or allow the connection as an *ingress* rule
- As a consequence: if a network policy restricts traffic going from/to a pod, the restriction cannot be overridden by a network policy selecting another pod
- This prevents an entity managing network policies in namespace A (but without permission to do so in namespace B) from adding network policies giving them access to namespace B

The rationale for network policies

- In network security, it is generally considered better to "deny all, then allow selectively"

  (The other approach, "allow all, then block selectively" makes it too easy to leave holes)

- As soon as one network policy selects a pod, the pod enters this "deny all" logic

- Further network policies can open additional access

- Good network policies should be scoped as precisely as possible

- In particular: make sure that the selector is not too broad

  (Otherwise, you end up affecting pods that were otherwise well secured)

Our first network policy

This is our game plan:

- run a web server in a pod
- create a network policy to block all access to the web server
- create another network policy to allow access only from specific pods

Running our test web server

Exercise

- Let's use the `nginx` image:

```
kubectl run testweb --image=nginx
```

- Find out the IP address of the pod with one of these two commands:

```
kubectl get pods -o wide -l run=testweb
IP=$(kubectl get pods -l run=testweb -o json | jq -r .items[0].status.podIP)
```

- Check that we can connect to the server:

```
curl $IP
```

The `curl` command should show us the "Welcome to nginx!" page.

Adding a very restrictive network policy

- The policy will select pods with the label `run=testweb`
- It will specify an empty list of ingress rules (matching nothing)

Exercise

- Apply the policy in this YAML file:

```
kubectl apply -f ~/container.training/k8s/netpol-deny-all-for-testweb.yaml
```

- Check if we can still access the server:

```
curl $IP
```

The `curl` command should now time out.

Looking at the network policy

This is the file that we applied:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-all-for-testweb
spec:
  podSelector:
    matchLabels:
      run: testweb
  ingress: []
```

Allowing connections only from specific pods

- We want to allow traffic from pods with the label `run=testcurl`

- Reminder: this label is automatically applied when we do `kubectl run testcurl ...`

Exercise

- Apply another policy:

```
kubectl apply -f ~/container.training/k8s/netpol-allow-testcurl-for-testweb.yaml
```

Looking at the network policy

This is the second file that we applied:

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-testcurl-for-testweb
spec:
  podSelector:
    matchLabels:
      run: testweb
  ingress:
  - from:
    - podSelector:
        matchLabels:
          run: testcurl
```

Testing the network policy

- Let's create pods with, and without, the required label

Exercise

- Try to connect to testweb from a pod with the `run=testcurl` label:

```
kubectl run testcurl --rm -i --image=centos -- curl -m3 $IP
```

- Try to connect to testweb with a different label:

```
kubectl run testkurl --rm -i --image=centos -- curl -m3 $IP
```

The first command will work (and show the "Welcome to nginx!" page).

The second command will fail and time out after 3 seconds.

(The timeout is obtained with the `-m3` option.)

An important warning

- Some network plugins only have partial support for network policies

- For instance, Weave doesn't support ipBlock (yet)

- Weave added support for egress rules in version 2.4 (released in July 2018)

- Unsupported features might be silently ignored

  (Making you believe that you are secure, when you're not)

Network policies, pods, and services

- Network policies apply to *pods*
- A *service* can select multiple pods

  (And load balance traffic across them)
- It is possible that we can connect to some pods, but not some others

  (Because of how network policies have been defined for these pods)
- In that case, connections to the service will randomly pass or fail

  (Depending on whether the connection was sent to a pod that we have access to or not)

Network policies and namespaces

- A good strategy is to isolate a namespace, so that:
  - all the pods in the namespace can communicate together
  - other namespaces cannot access the pods
  - external access has to be enabled explicitly
- Let's see what this would look like for the DockerCoins app!

Network policies for DockerCoins

- We are going to apply two policies
- The first policy will prevent traffic from other namespaces
- The second policy will allow traffic to the `webui` pods
- That's all we need for that app!

Blocking traffic from other namespaces

This policy selects all pods in the current namespace.

It allows traffic only from pods in the current namespace.

(An empty `podSelector` means "all pods".)

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-from-other-namespaces
spec:
  podSelector: {}
  ingress:
  - from:
    - podSelector: {}
```

Allowing traffic to `webui` pods

This policy selects all pods with label `run=webui`.

It allows traffic from any source.

(An empty `from` fields means "all sources".)

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-webui
spec:
  podSelector:
    matchLabels:
      run: webui
  ingress:
  - from: []
```

Applying both network policies

- Both network policies are declared in the file `k8s/netpol-dockercoins.yaml`

Exercise

- Apply the network policies:

```
kubectl apply -f ~/container.training/k8s/netpol-dockercoins.yaml
```

- Check that we can still access the web UI from outside
  (and that the app is still working correctly!)

- Check that we can't connect anymore to `rng` or `hasher` through their ClusterIP

Note: using `kubectl proxy` or `kubectl port-forward` allows us to connect regardless of existing network policies. This allows us to debug and troubleshoot easily, without having to poke holes in our firewall.

Cleaning up our network policies

- The network policies that we have installed block all traffic to the default namespace

- We should remove them, otherwise further exercises will fail!

Exercise

- Remove all network policies:

```
kubectl delete networkpolicies --all
```

Protecting the control plane

- Should we add network policies to block unauthorized access to the control plane?

  (etcd, API server, etc.)

Protecting the control plane

- Should we add network policies to block unauthorized access to the control plane?

  (etcd, API server, etc.)

- At first, it seems like a good idea ...

Protecting the control plane

- Should we add network policies to block unauthorized access to the control plane?

  (etcd, API server, etc.)

- At first, it seems like a good idea ...

- But it *shouldn't* be necessary:

  - not all network plugins support network policies

  - the control plane is secured by other methods (mutual TLS, mostly)

  - the code running in our pods can reasonably expect to contact the API
(and it can do so safely thanks to the API permission model)

- If we block access to the control plane, we might disrupt legitimate code

- ... Without necessarily improving security

Further resources

- As always, the Kubernetes documentation is a good starting point
- The API documentation has a lot of detail about the format of various objects:
  - NetworkPolicy
  - NetworkPolicySpec
  - NetworkPolicyIngressRule
  - etc.
- And two resources by Ahmet Alp Balkan:
  - a very good talk about network policies at KubeCon North America 2017
  - a repository of ready-to-use recipes for network policies


ECS
European Containers
ECS
EuropeanContainers.com

Authentication and authorization

Previous section | Back to table of contents | Next section

Authentication and authorization

And first, a little refresher!

- Authentication = verifying the identity of a person

  On a UNIX system, we can authenticate with login+password, SSH keys ...

- Authorization = listing what they are allowed to do

  On a UNIX system, this can include file permissions, sudoer entries ...

- Sometimes abbreviated as "authn" and "authz"

- In good modular systems, these things are decoupled

  (so we can e.g. change a password or SSH key without having to reset access rights)

Authentication in Kubernetes

- When the API server receives a request, it tries to authenticate it

  (it examines headers, certificates ... anything available)

- Many authentication methods are available and can be used simultaneously

  (we will see them on the next slide)

- It's the job of the authentication method to produce:

  - the user name
  - the user ID
  - a list of groups

- The API server doesn't interpret these; it'll be the job of *authorizers*

Authentication methods

- TLS client certificates

  (that's what we've been doing with `kubectl` so far)

- Bearer tokens

  (a secret token in the HTTP headers of the request)

- HTTP basic auth

  (carrying user and password in a HTTP header)

- Authentication proxy

  (sitting in front of the API and setting trusted headers)

Anonymous requests

- If any authentication method *rejects* a request, it's denied

  (`401 Unauthorized` HTTP code)

- If a request is neither accepted nor accepted by anyone, it's anonymous

  - the user name is `system:anonymous`

  - the list of groups is `[system:unauthenticated]`

- By default, the anonymous user can't do anything

  (that's what you get if you just `curl` the Kubernetes API)

Authentication with TLS certificates

- This is enabled in most Kubernetes deployments
- The user name is derived from the `CN` in the client certificates
- The groups are derived from the `O` fields in the client certificate
- From the point of view of the Kubernetes API, users do not exist

  (i.e. they are not stored in etcd or anywhere else)
- Users can be created (and given membership to groups) independently of the API
- The Kubernetes API can be set up to use your custom CA to validate client certs

Viewing our admin certificate

- Let's inspect the certificate we've been using all this time!

Exercise

- This command will show the `CN` and `O` fields for our certificate:

```
kubectl config view \
      --raw \
      -o json \
      | jq -r .users[0].user[\"client-certificate-data\"] \
      | base64 -d \
      | openssl x509 -text \
      | grep Subject:
```

Let's break down that command together! 😅

Breaking down the command

- `kubectl config view` shows the Kubernetes user configuration
- `--raw` includes certificate information (which shows as REDACTED otherwise)
- `-o json` outputs the information in JSON format
- `| jq ...` extracts the field with the user certificate (in base64)
- `| base64 -d` decodes the base64 format (now we have a PEM file)
- `| openssl x509 -text` parses the certificate and outputs it as plain text
- `| grep Subject:` shows us the line that interests us

→ We are user `kubernetes-admin`, in group `system:masters`.

User certificates in practice

- The Kubernetes API server does not support certificate revocation

  (see issue #18982)

- As a result, we cannot easily suspend a user's access

- There are workarounds, but they are very inconvenient:

  - issue short-lived certificates (e.g. 24 hours) and regenerate them often

  - re-create the CA and re-issue all certificates in case of compromise

  - grant permissions to individual users, not groups
    (and remove all permissions to a compromised user)

- Until this is fixed, we probably want to use other methods

Authentication with tokens

- Tokens are passed as HTTP headers:

  `Authorization: Bearer and-then-here-comes-the-token`

- Tokens can be validated through a number of different methods:

  - static tokens hard-coded in a file on the API server

  - bootstrap tokens (special case to create a cluster or join nodes)

  - OpenID Connect tokens (to delegate authentication to compatible OAuth2 providers)

  - service accounts (these deserve more details, coming right up!)

Service accounts

- A service account is a user that exists in the Kubernetes API

  (it is visible with e.g. `kubectl get serviceaccounts`)

- Service accounts can therefore be created / updated dynamically

  (they don't require hand-editing a file and restarting the API server)

- A service account is associated with a set of secrets

  (the kind that you can view with `kubectl get secrets`)

- Service accounts are generally used to grant permissions to applications, services ...

  (as opposed to humans)

Token authentication in practice

- We are going to list existing service accounts

- Then we will extract the token for a given service account

- And we will use that token to authenticate with the API

Listing service accounts

Exercise

- The resource name is `serviceaccount` or `sa` in short:

```
kubectl get sa
```

There should be just one service account in the default namespace: `default`.

Finding the secret

Exercise

- List the secrets for the `default` service account:

```
kubectl get sa default -o yaml
SECRET=$(kubectl get sa default -o json | jq -r .secrets[0].name)
```

It should be named `default-token-XXXXX`.

Extracting the token

- The token is stored in the secret, wrapped with base64 encoding

Exercise

- View the secret:

```
kubectl get secret $SECRET -o yaml
```

- Extract the token and decode it:

```
TOKEN=$(kubectl get secret $SECRET -o json \
      | jq -r .data.token | base64 -d)
```

Using the token

- Let's send a request to the API, without and with the token

Exercise

- Find the ClusterIP for the `kubernetes` service:

```
kubectl get svc kubernetes
API=$(kubectl get svc kubernetes -o json | jq -r .spec.clusterIP)
```

- Connect without the token:

```
curl -k https://$API
```

- Connect with the token:

```
curl -k -H "Authorization: Bearer $TOKEN" https://$API
```

Results

- In both cases, we will get a "Forbidden" error

- Without authentication, the user is `system:anonymous`

- With authentication, it is shown as `system:serviceaccount:default:default`

- The API "sees" us as a different user

- But neither user has any right, so we can't do nothin'

- Let's change that!

Authorization in Kubernetes

- There are multiple ways to grant permissions in Kubernetes, called authorizers:
  - Node Authorization (used internally by kubelet; we can ignore it)
  - Attribute-based access control (powerful but complex and static; ignore it too)
  - Webhook (each API request is submitted to an external service for approval)
  - Role-based access control (associates permissions to users dynamically)
- The one we want is the last one, generally abbreviated as RBAC

Role-based access control

- RBAC allows to specify fine-grained permissions

- Permissions are expressed as *rules*

- A rule is a combination of:

  - verbs like create, get, list, update, delete ...

  - resources (as in "API resource", like pods, nodes, services ...)

  - resource names (to specify e.g. one specific pod instead of all pods)

  - in some case, subresources (e.g. logs are subresources of pods)

From rules to roles to rolebindings

- A *role* is an API object containing a list of *rules*

  Example: role "external-load-balancer-configurator" can:

  - [list, get] resources [endpoints, services, pods]
  - [update] resources [services]

- A *rolebinding* associates a role with a user

  Example: rolebinding "external-load-balancer-configurator":

  - associates user "external-load-balancer-configurator"
  - with role "external-load-balancer-configurator"

- Yes, there can be users, roles, and rolebindings with the same name

- It's a good idea for 1-1-1 bindings; not so much for 1-N ones

Cluster-scope permissions

- API resources Role and RoleBinding are for objects within a namespace
- We can also define API resources ClusterRole and ClusterRoleBinding
- These are a superset, allowing to:
  - specify actions on cluster-wide objects (like nodes)
  - operate across all namespaces
- We can create Role and RoleBinding resources within a namespaces
- ClusterRole and ClusterRoleBinding resources are global

Pods and service accounts

- A pod can be associated to a service account
  - by default, it is associated to the `default` service account
  - as we've seen earlier, this service account has no permission anyway
- The associated token is exposed into the pod's filesystem

  (in `/var/run/secrets/kubernetes.io/serviceaccount/token`)
- Standard Kubernetes tooling (like `kubectl`) will look for it there
- So Kubernetes tools running in a pod will automatically use the service account

In practice

- We are going to create a service account
- We will use an existing cluster role (`view`)
- We will bind together this role and this service account
- Then we will run a pod using that service account
- In this pod, we will install `kubectl` and check our permissions

Creating a service account

- We will call the new service account `viewer`

  (note that nothing prevents us from calling it `view`, like the role)

Exercise

- Create the new service account:

```
kubectl create serviceaccount viewer
```

- List service accounts now:

```
kubectl get serviceaccounts
```

Binding a role to the service account

- Binding a role = creating a *rolebinding* object
- We will call that object `viewercanview`

  (but again, we could call it `view`)

Exercise

- Create the new role binding:

```
kubectl create rolebinding viewercanview \
      --clusterrole=view \
      --serviceaccount=default:viewer
```

It's important to note a couple of details in these flags ...

Roles vs Cluster Roles

- We used `--clusterrole=view`
- What would have happened if we had used `--role=view`?
  - we would have bound the role `view` from the local namespace (instead of the cluster role `view`)
  - the command would have worked fine (no error)
  - but later, our API requests would have been denied
- This is a deliberate design decision

  (we can reference roles that don't exist, and create/update them later)

Users vs Service Accounts

- We used `--serviceaccount=default:viewer`
- What would have happened if we had used `--user=default:viewer`?
  - we would have bound the role to a user instead of a service account
  - again, the command would have worked fine (no error)
  - ... but our API requests would have been denied later
- What's about the `default:` prefix?
  - that's the namespace of the service account
  - yes, it could be inferred from context, but ... `kubectl` requires it

Testing

- We will run an `alpine` pod and install `kubectl` there

Exercise

- Run a one-time pod:

```
kubectl run eyepod --rm -ti --restart=Never \
      --serviceaccount=viewer \
      --image alpine
```

- Install `curl`, then use it to install `kubectl`:

```
apk add --no-cache curl
URLBASE=https://storage.googleapis.com/kubernetes-release/release
KUBEVER=$(curl -s $URLBASE/stable.txt)
curl -LO $URLBASE/$KUBEVER/bin/linux/amd64/kubectl
chmod +x kubectl
```

Running `kubectl` in the pod

- We'll try to use our `view` permissions, then to create an object

Exercise

- Check that we can, indeed, view things:

```
./kubectl get all
```

- But that we can't create things:

```
./kubectl run tryme --image=nginx
```

- Exit the container with `exit` or `^D`

Testing directly with `kubectl`

- We can also check for permission with `kubectl auth can-i`:

```
kubectl auth can-i list nodes
kubectl auth can-i create pods
kubectl auth can-i get pod/name-of-pod
kubectl auth can-i get /url-fragment-of-api-request/
kubectl auth can-i '*' services
```

- And we can check permissions on behalf of other users:

```
kubectl auth can-i list nodes \
      --as some-user
kubectl auth can-i list nodes \
      --as system:serviceaccount:<namespace>:<name-of-service-account>
```

Exposing HTTP services with Ingress resources

Previous section | Back to table of contents | Next section

Exposing HTTP services with Ingress resources

- *Services* give us a way to access a pod or a set of pods
- Services can be exposed to the outside world:
  - with type `NodePort` (on a port >30000)
  - with type `LoadBalancer` (allocating an external load balancer)
- What about HTTP services?
  - how can we expose `webui`, `rng`, `hasher`?
  - the Kubernetes dashboard?
  - a new version of `webui`?

Exposing HTTP services

- If we use `NodePort` services, clients have to specify port numbers

  (i.e. http://xxxxx:31234 instead of just http://xxxxx)

- `LoadBalancer` services are nice, but:

  - they are not available in all environments

  - they often carry an additional cost (e.g. they provision an ELB)

  - they require one extra step for DNS integration
    (waiting for the `LoadBalancer` to be provisioned; then adding it to DNS)

- We could build our own reverse proxy

Building a custom reverse proxy

- There are many options available:

  Apache, HAProxy, Hipache, NGINX, Traefik, ...

  (look at jpetazzo/aiguillage for a minimal reverse proxy configuration using NGINX)

- Most of these options require us to update/edit configuration files after each change

- Some of them can pick up virtual hosts and backends from a configuration store

- Wouldn't it be nice if this configuration could be managed with the Kubernetes API?

Building a custom reverse proxy

- There are many options available:

  Apache, HAProxy, Hipache, NGINX, Traefik, ...

  (look at jpetazzo/aiguillage for a minimal reverse proxy configuration using NGINX)

- Most of these options require us to update/edit configuration files after each change

- Some of them can pick up virtual hosts and backends from a configuration store

- Wouldn't it be nice if this configuration could be managed with the Kubernetes API?

- Enter[1] *Ingress* resources!

[1] Pun maybe intended.

Ingress resources

- Kubernetes API resource (`kubectl get ingress`/`ingresses`/`ing`)
- Designed to expose HTTP services
- Basic features:
  - load balancing
  - SSL termination
  - name-based virtual hosting
- Can also route to different services depending on:
  - URI path (e.g. `/api`→`api-service`, `/static`→`assets-service`)
  - Client headers, including cookies (for A/B testing, canary deployment…)
  - and more!

Principle of operation

- Step 1: deploy an *ingress controller*
  - ingress controller = load balancer + control loop
  - the control loop watches over ingress resources, and configures the LB accordingly
- Step 2: setup DNS
  - associate DNS entries with the load balancer address
- Step 3: create *ingress resources*
  - the ingress controller picks up these resources and configures the LB
- Step 4: profit!

Ingress in action

- We will deploy the Traefik ingress controller
  - this is an arbitrary choice
  - maybe motivated by the fact that Traefik releases are named after cheeses
- For DNS, we will use nip.io
  - `*.1.2.3.4.nip.io` resolves to `1.2.3.4`
- We will create ingress resources for various HTTP services

Deploying pods listening on port 80

- We want our ingress load balancer to be available on port 80
- We could do that with a `LoadBalancer` service

  ... but it requires support from the underlying infrastructure
- We could use pods specifying `hostPort: 80`

  ... but with most CNI plugins, this doesn't work or require additional setup
- We could use a `NodePort` service

  ... but that requires changing the `--service-node-port-range` flag in the API server
- Last resort: the `hostNetwork` mode

Without `hostNetwork`

- Normally, each pod gets its own *network namespace*

  (sometimes called sandbox or network sandbox)

- An IP address is associated to the pod

- This IP address is routed/connected to the cluster network

- All containers of that pod are sharing that network namespace

  (and therefore using the same IP address)

With `hostNetwork: true`

- No network namespace gets created
- The pod is using the network namespace of the host
- It "sees" (and can use) the interfaces (and IP addresses) of the host
- The pod can receive outside traffic directly, on any port
- Downside: with most network plugins, network policies won't work for that pod
  - most network policies work at the IP address level
  - filtering that pod = filtering traffic from the node

Running Traefik

- The Traefik documentation tells us to pick between Deployment and Daemon Set
- We are going to use a Daemon Set so that each node can accept connections
- We will do two minor changes to the YAML provided by Traefik:
  - enable `hostNetwork`
  - add a *toleration* so that Traefik also runs on `node1`

Taints and tolerations

- A *taint* is an attribute added to a node
- It prevents pods from running on the node
- ... Unless they have a matching *toleration*
- When deploying with `kubeadm`:
  - a taint is placed on the node dedicated the control plane
  - the pods running the control plane have a matching toleration

Checking taints on our nodes

Exercise

- Check our nodes specs:

```
kubectl get node node1 -o json | jq .spec
kubectl get node node2 -o json | jq .spec
```

We should see a result only for `node1` (the one with the control plane):

```
"taints": [
  {
    "effect": "NoSchedule",
    "key": "node-role.kubernetes.io/master"
  }
]
```

Understanding a taint

- The `key` can be interpreted as:

  - a reservation for a special set of pods
    (here, this means "this node is reserved for the control plane")

  - an error condition on the node
    (for instance: "disk full", do not start new pods here!)

- The `effect` can be:

  - `NoSchedule` (don't run new pods here)

  - `PreferNoSchedule` (try not to run new pods here)

  - `NoExecute` (don't run new pods and evict running pods)

Checking tolerations on the control plane

Exercise

- Check tolerations for CoreDNS:

```
kubectl -n kube-system get deployments coredns -o json |
      jq .spec.template.spec.tolerations
```

The result should include:

```
{
  "effect": "NoSchedule",
  "key": "node-role.kubernetes.io/master"
}
```

It means: "bypass the exact taint that we saw earlier on `node1`."

Special tolerations

Exercise

- Check tolerations on `kube-proxy`:

```
kubectl -n kube-system get ds kube-proxy -o json |
      jq .spec.template.spec.tolerations
```

The result should include:

```
{
  "operator": "Exists"
}
```

This one is a special case that means "ignore all taints and run anyway."

Running Traefik on our cluster

- We provide a YAML file (`k8s/traefik.yaml`) which is essentially the sum of:
  - Traefik's Daemon Set resources (patched with `hostNetwork` and tolerations)
  - Traefik's RBAC rules allowing it to watch necessary API objects

Exercise

- Apply the YAML:

```
kubectl apply -f ~/container.training/k8s/traefik.yaml
```

Checking that Traefik runs correctly

- If Traefik started correctly, we now have a web server listening on each node

Exercise

- Check that Traefik is serving 80/tcp:

```
curl localhost
```

We should get a `404 page not found` error.

This is normal: we haven't provided any ingress rule yet.

Setting up DNS

- To make our lives easier, we will use nip.io
- Check out `http://cheddar.A.B.C.D.nip.io`

  (replacing A.B.C.D with the IP address of `node1`)
- We should get the same `404 page not found` error

  (meaning that our DNS is "set up properly", so to speak!)

Traefik web UI

- Traefik provides a web dashboard

- With the current install method, it's listening on port 8080

Exercise

- Go to `http://node1:8080` (replacing `node1` with its IP address)

Setting up host-based routing ingress rules

- We are going to use `errm/cheese` images

  (there are 3 tags available: wensleydale, cheddar, stilton)

- These images contain a simple static HTTP server sending a picture of cheese

- We will run 3 deployments (one for each cheese)

- We will create 3 services (one for each deployment)

- Then we will create 3 ingress rules (one for each service)

- We will route `<name-of-cheese>.A.B.C.D.nip.io` to the corresponding deployment

Running cheesy web servers

Exercise

- Run all three deployments:

```
kubectl run cheddar --image=errm/cheese:cheddar
kubectl run stilton --image=errm/cheese:stilton
kubectl run wensleydale --image=errm/cheese:wensleydale
```

- Create a service for each of them:

```
kubectl expose deployment cheddar --port=80
kubectl expose deployment stilton --port=80
kubectl expose deployment wensleydale --port=80
```

What does an ingress resource look like?

Here is a minimal host-based ingress resource:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: cheddar
spec:
  rules:
  - host: cheddar.A.B.C.D.nip.io
    http:
      paths:
      - path: /
        backend:
          serviceName: cheddar
          servicePort: 80
```

(It is in `k8s/ingress.yaml`.)

Creating our first ingress resources

Exercise

- Edit the file `~/container.training/k8s/ingress.yaml`
- Replace A.B.C.D with the IP address of `node1`
- Apply the file
- Open http://cheddar.A.B.C.D.nip.io

(An image of a piece of cheese should show up.)

Creating the other ingress resources

Exercise

- Edit the file `~/container.training/k8s/ingress.yaml`
- Replace `cheddar` with `stilton` (in `name`, `host`, `serviceName`)
- Apply the file
- Check that `stilton.A.B.C.D.nip.io` works correctly
- Repeat for `wensleydale`

Using multiple ingress controllers

- You can have multiple ingress controllers active simultaneously

  (e.g. Traefik and NGINX)

- You can even have multiple instances of the same controller

  (e.g. one for internal, another for external traffic)

- The `kubernetes.io/ingress.class` annotation can be used to tell which one to use

- It's OK if multiple ingress controllers configure the same resource

  (it just means that the service will be accessible through multiple paths)

Ingress: the good

- The traffic flows directly from the ingress load balancer to the backends
  - it doesn't need to go through the `ClusterIP`
  - in fact, we don't even need a `ClusterIP` (we can use a headless service)
- The load balancer can be outside of Kubernetes

  (as long as it has access to the cluster subnet)
- This allows to use external (hardware, physical machines...) load balancers
- Annotations can encode special features

  (rate-limiting, A/B testing, session stickiness, etc.)

Ingress: the bad

- Aforementioned "special features" are not standardized yet
- Some controllers will support them; some won't
- Even relatively common features (stripping a path prefix) can differ:
  - traefik.ingress.kubernetes.io/rule-type: PathPrefixStrip
  - ingress.kubernetes.io/rewrite-target: /
- This should eventually stabilize

  (remember that ingresses are currently `apiVersion: extensions/v1beta1`)

Collecting metrics with Prometheus

Collecting metrics with Prometheus

- Prometheus is an open-source monitoring system including:
  - multiple *service discovery* backends to figure out which metrics to collect
  - a *scraper* to collect these metrics
  - an efficient *time series database* to store these metrics
  - a specific query language (PromQL) to query these time series
  - an *alert manager* to notify us according to metrics values or trends
- We are going to deploy it on our Kubernetes cluster and see how to query it

Why Prometheus?

- We don't endorse Prometheus more or less than any other system
- It's relatively well integrated within the Cloud Native ecosystem
- It can be self-hosted (this is useful for tutorials like this)
- It can be used for deployments of varying complexity:
  - one binary and 10 lines of configuration to get started
  - all the way to thousands of nodes and millions of metrics

Exposing metrics to Prometheus

- Prometheus obtains metrics and their values by querying *exporters*
- An exporter serves metrics over HTTP, in plain text
- 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
- If you want to expose custom metrics to Prometheus:
  - serve a text page like these, and you're good to go
  - libraries are available in various languages to help with quantiles etc.

How Prometheus gets these metrics

- The *Prometheus server* will *scrape* URLs like these at regular intervals

  (by default: every minute; can be more/less frequent)

- If you're worried about parsing overhead: exporters can also use protobuf

- The list of URLs to scrape (the *scrape targets*) is defined in configuration

Defining scrape targets

This is maybe the simplest configuration file for Prometheus:

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
```

- In this configuration, Prometheus collects its own internal metrics
- A typical configuration file will have multiple `scrape_configs`
- In this configuration, the list of targets is fixed
- A typical configuration file will use dynamic service discovery

Service discovery

This configuration file will leverage existing DNS `A` records:

```
scrape_configs:
  - ...
  - job_name: 'node'
    dns_sd_configs:
      - names: ['api-backends.dc-paris-2.enix.io']
        type: 'A'
        port: 9100
```

- In this configuration, Prometheus resolves the provided name(s)

  (here, `api-backends.dc-paris-2.enix.io`)

- Each resulting IP address is added as a target on port 9100

Dynamic service discovery

- In the DNS example, the names are re-resolved at regular intervals
- As DNS records are created/updated/removed, scrape targets change as well
- Existing data (previously collected metrics) is not deleted
- Other service discovery backends work in a similar fashion

Other service discovery mechanisms

- Prometheus can connect to e.g. a cloud API to list instances
- Or to the Kubernetes API to list nodes, pods, services ...
- Or a service like Consul, Zookeeper, etcd, to list applications
- The resulting configurations files are *way more complex*

  (but don't worry, we won't need to write them ourselves)

Time series database

- We could wonder, "why do we need a specialized database?"
- One metrics data point = metrics ID + timestamp + value
- With a classic SQL or noSQL data store, that's at least 160 bits of data + indexes
- Prometheus is way more efficient, without sacrificing performance

  (it will even be gentler on the I/O subsystem since it needs to write less)

Storage in Prometheus 2.0 by Goutham V at DC17EU

Running Prometheus on our cluster

We need to:

- Run the Prometheus server in a pod

  (using e.g. a Deployment to ensure that it keeps running)

- Expose the Prometheus server web UI (e.g. with a NodePort)

- Run the *node exporter* on each node (with a Daemon Set)

- Setup a Service Account so that Prometheus can query the Kubernetes API

- Configure the Prometheus server

  (storing the configuration in a Config Map for easy updates)

Helm Charts to the rescue

- To make our lives easier, we are going to use a Helm Chart
- The Helm Chart will take care of all the steps explained above

  (including some extra features that we don't need, but won't hurt)

Step 1: install Helm

- If we already installed Helm earlier, these commands won't break anything

- Install Tiller (Helm's server-side component) on our cluster:

```
helm init
```

- Give Tiller permission to deploy things on our cluster:

```
kubectl create clusterrolebinding add-on-cluster-admin \
  --clusterrole=cluster-admin --serviceaccount=kube-system:default
```

Step 2: install Prometheus

- Skip this if we already installed Prometheus earlier

  (in doubt, check with `helm list`)

- Install Prometheus on our cluster:

```
helm install stable/prometheus \
     --set server.service.type=NodePort \
     --set server.persistentVolume.enabled=false
```

The provided flags:

- expose the server web UI (and API) on a NodePort

- use an ephemeral volume for metrics storage
  (instead of requesting a Persistent Volume through a Persistent Volume Claim)

Connecting to the Prometheus web UI

- Let's connect to the web UI and see what we can do

Exercise

- Figure out the NodePort that was allocated to the Prometheus server:

```
kubectl get svc | grep prometheus-server
```

- With your browser, connect to that port

Querying some metrics

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

Exercise

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

```
sum by (instance) (
  irate(
    container_cpu_usage_seconds_total{
      pod_name=~"worker.*"
      }[5m]
  )
)
```

- Click on the blue "Execute" button and on the "Graph" tab just below
- We see the cumulated CPU usage of worker pods for each node
  (if we just deployed Prometheus, there won't be much data to see, though)

Getting started with PromQL

- We can't learn PromQL in just 5 minutes
- But we can cover the basics to get an idea of what is possible

  (and have some keywords and pointers)
- We are going to break down the query above

  (building it one step at a time)

Graphing one metric across all tags

This query will show us CPU usage across all containers:

```
container_cpu_usage_seconds_total
```

- The suffix of the metrics name tells us:
  - the unit (seconds of CPU)
  - that it's the total used since the container creation
- Since it's a "total", it is an increasing quantity

  (we need to compute the derivative if we want e.g. CPU % over time)
- We see that the metrics retrieved have *tags* attached to them

Selecting metrics with tags

This query will show us only metrics for worker containers:

```
container_cpu_usage_seconds_total{pod_name=~"worker.*"}
```

- The `=~` operator allows regex matching

- We select all the pods with a name starting with `worker`

  (it would be better to use labels to select pods; more on that later)

- The result is a smaller set of containers

Transforming counters in rates

This query will show us CPU usage % instead of total seconds used:

```
100*irate(container_cpu_usage_seconds_total{pod_name=~"worker.*"}[5m])
```

- The `irate` operator computes the "per-second instant rate of increase"
  - `rate` is similar but allows decreasing counters and negative values
  - with `irate`, if a counter goes back to zero, we don't get a negative spike
- The `[5m]` tells how far to look back if there is a gap in the data
- And we multiply with `100*` to get CPU % usage

Aggregation operators

This query sums the CPU usage per node:

```
sum by (instance) (
  irate(container_cpu_usage_seconds_total{pod_name=~"worker.*"}[5m])
)
```

- `instance` corresponds to the node on which the container is running
- `sum by (instance) (...)` computes the sum for each instance
- Note: all the other tags are collapsed

  (in other words, the resulting graph only shows the `instance` tag)
- PromQL supports many more aggregation operators

What kind of metrics can we collect?

- Node metrics (related to physical or virtual machines)

- Container metrics (resource usage per container)

- Databases, message queues, load balancers, ...

  (check out this list of exporters!)

- Instrumentation (=deluxe `printf` for our code)

- Business metrics (customers served, revenue, ...)

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/

Application metrics

- Arbitrary metrics related to your application and business
- System performance: request latency, error rate ...
- Volume information: number of rows in database, message queue size ...
- Business data: inventory, items sold, revenue ...

Detecting scrape targets

- Prometheus can leverage Kubernetes service discovery

  (with proper configuration)

- Services or pods can be annotated with:

  - `prometheus.io/scrape: true` to enable scraping
  - `prometheus.io/port: 9090` to indicate the port number
  - `prometheus.io/path: /metrics` to indicate the URI (`/metrics` by default)

- Prometheus will detect and scrape these (without needing a restart or reload)

Querying labels

- What if we want to get metrics for containers belong to pod tagged `worker`?
- The cAdvisor exporter does not give us Kubernetes labels
- Kubernetes labels are exposed through another exporter
- We can see Kubernetes labels through metrics `kube_pod_labels`

  (each container appears as a time series with constant value of `1`)
- Prometheus *kind of* supports "joins" between time series
- But only if the names of the tags match exactly

Unfortunately ...

- The cAdvisor exporter uses tag `pod_name` for the name of a pod
- The Kubernetes service endpoints exporter uses tag `pod` instead
- And this is why we can't have nice things
- See Prometheus issue #2204 for the rationale

  (this comment in particular if you want a workaround involving relabeling)
- Then see this blog post or this other one to see how to perform "joins"
- There is a good chance that the situation will improve in the future


한진 보스톤
HANJIN BOSTON
MAJURO

Volumes

Volumes

- Volumes are special directories that are mounted in containers
- Volumes can have many different purposes:
  - share files and directories between containers running on the same machine
  - share files and directories between containers and their host
  - centralize configuration information in Kubernetes and expose it to containers
  - manage credentials and secrets and expose them securely to containers
  - store persistent data for stateful services
  - access storage systems (like Ceph, EBS, NFS, Portworx, and many others)

Kubernetes volumes vs. Docker volumes

- Kubernetes and Docker volumes are very similar

  (the Kubernetes documentation says otherwise ...
  but it refers to Docker 1.7, which was released in 2015!)

- Docker volumes allow to share data between containers running on the same host

- Kubernetes volumes allow us to share data between containers in the same pod

- Both Docker and Kubernetes volumes allow us access to storage systems

- Kubernetes volumes are also used to expose configuration and secrets

- Docker has specific concepts for configuration and secrets

  (but under the hood, the technical implementation is similar)

- If you're not familiar with Docker volumes, you can safely ignore this slide!

A simple volume example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-volume
spec:
  volumes:
  - name: www
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html/
```

A simple volume example, explained

- We define a standalone `Pod` named `nginx-with-volume`
- In that pod, there is a volume named `www`
- No type is specified, so it will default to `emptyDir`

  (as the name implies, it will be initialized as an empty directory at pod creation)
- In that pod, there is also a container named `nginx`
- That container mounts the volume `www` to path `/usr/share/nginx/html/`

A volume shared between two containers

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-volume
spec:
  volumes:
  - name: www
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html/
  - name: git
    image: alpine
    command: [ "sh", "-c", "apk add --no-cache git && git clone https://github.com/octocat/Spoon-Knife /www" ]
    volumeMounts:
    - name: www
      mountPath: /www/
  restartPolicy: OnFailure
```

Sharing a volume, explained

- We added another container to the pod

- That container mounts the `www` volume on a different path (`/www`)

- It uses the `alpine` image

- When started, it installs `git` and clones the `octocat/Spoon-Knife` repository

  (that repository contains a tiny HTML website)

- As a result, NGINX now serves this website

Sharing a volume, in action

- Let's try it!

Exercise

- Create the pod by applying the YAML file:

```
kubectl apply -f ~/container.training/k8s/nginx-with-volume.yaml
```

- Check the IP address that was allocated to our pod:

```
kubectl get pod nginx-with-volume -o wide
IP=$(kubectl get pod nginx-with-volume -o json | jq -r .status.podIP)
```

- Access the web server:

```
curl $IP
```

The devil is in the details

- The default `restartPolicy` is `Always`
- This would cause our `git` container to run again … and again … and again

  (with an exponential back-off delay, as explained in the documentation)
- That's why we specified `restartPolicy: OnFailure`
- There is a short period of time during which the website is not available

  (because the `git` container hasn't done its job yet)
- This could be avoided by using Init Containers

  (we will see a live example in a few sections)

Volume lifecycle

- The lifecycle of a volume is linked to the pod's lifecycle
- This means that a volume is created when the pod is created
- This is mostly relevant for `emptyDir` volumes

  (other volumes, like remote storage, are not "created" but rather "attached")
- A volume survives across container restarts
- A volume is destroyed (or, for remote storage, detached) when the pod is destroyed

Building images with the Docker Engine

Building images with the Docker Engine

- Until now, we have built our images manually, directly on a node
- We are going to show how to build images from within the cluster

  (by executing code in a container controlled by Kubernetes)
- We are going to use the Docker Engine for that purpose
- To access the Docker Engine, we will mount the Docker socket in our container
- After building the image, we will push it to our self-hosted registry

Resource specification for our builder pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: build-image
spec:
  restartPolicy: OnFailure
  containers:
  - name: docker-build
    image: docker
    env:
    - name: REGISTRY_PORT
      value: "3XXXX"
    command: ["sh", "-c"]
    args:
    - |
      apk add --no-cache git &&
      mkdir /workspace &&
      git clone https://github.com/jpetazzo/container.training /workspace &&
      docker build -t localhost:$REGISTRY_PORT/worker /workspace/dockercoins/worker &&
      docker push localhost:$REGISTRY_PORT/worker
    volumeMounts:
    - name: docker-socket
      mountPath: /var/run/docker.sock
  volumes:
  - name: docker-socket
    hostPath:
      path: /var/run/docker.sock
```

Breaking down the pod specification (1/2)

- `restartPolicy: OnFailure` prevents the build from running in an infinite lopo
- We use the `docker` image (so that the `docker` CLI is available)
- We rely on the fact that the `docker` image is based on `alpine`

  (which is why we use `apk` to install `git`)
- The port for the registry is passed through an environment variable

  (this avoids repeating it in the specification, which would be error-prone)

⚠️ The environment variable has to be a string, so the `"`s are mandatory!

Breaking down the pod specification (2/2)

- The volume `docker-socket` is declared with a `hostPath`, indicating a bind-mount
- It is then mounted in the container onto the default Docker socket path
- We show a interesting way to specify the commands to run in the container:
  - the command executed will be `sh -c <args>`
  - `args` is a list of strings
  - `|` is used to pass a multi-line string in the YAML file

Running our pod

- Let's try this out!

Exercise

- Check the port used by our self-hosted registry:

```
kubectl get svc registry
```

- Edit `~/container.training/k8s/docker-build.yaml` to put the port number

- Schedule the pod by applying the resource file:

```
kubectl apply -f ~/container.training/k8s/docker-build.yaml
```

- Watch the logs:

```
stern build-image
```

What's missing?

What do we need to change to make this production-ready?

- Build from a long-running container (e.g. a `Deployment`) triggered by web hooks

  (the payload of the web hook could indicate the repository to build)

- Build a specific branch or tag; tag image accordingly

- Handle repositories where the Dockerfile is not at the root

  (or containing multiple Dockerfiles)

- Expose build logs so that troubleshooting is straightforward

What's missing?

What do we need to change to make this production-ready?

- Build from a long-running container (e.g. a `Deployment`) triggered by web hooks

  (the payload of the web hook could indicate the repository to build)

- Build a specific branch or tag; tag image accordingly

- Handle repositories where the Dockerfile is not at the root

  (or containing multiple Dockerfiles)

- Expose build logs so that troubleshooting is straightforward

🤔 That seems like a lot of work!

What's missing?

What do we need to change to make this production-ready?

- Build from a long-running container (e.g. a `Deployment`) triggered by web hooks

  (the payload of the web hook could indicate the repository to build)

- Build a specific branch or tag; tag image accordingly

- Handle repositories where the Dockerfile is not at the root

  (or containing multiple Dockerfiles)

- Expose build logs so that troubleshooting is straightforward

🤔 That seems like a lot of work!

That's why services like Docker Hub (with automated builds) are helpful.
They handle the whole "code repository → Docker image" workflow.

Things to be aware of

- This is talking directly to a node's Docker Engine to build images

- It bypasses resource allocation mechanisms used by Kubernetes

  (but you can use *taints* and *tolerations* to dedicate builder nodes)

- Be careful not to introduce conflicts when naming images

  (e.g. do not allow the user to specify the image names!)

- Your builds are going to be *fast*

  (because they will leverage Docker's caching system)


© Lode Van de Velde

Building images with Kaniko

Previous section | Back to table of contents | Next section

Building images with Kaniko

- Kaniko is an open source tool to build container images within Kubernetes

- It can build an image using any standard Dockerfile

- The resulting image can be pushed to a registry or exported as a tarball

- It doesn't require any particular privilege

  (and can therefore run in a regular container in a regular pod)

- This combination of features is pretty unique

  (most other tools use different formats, or require elevated privileges)

Kaniko in practice

- Kaniko provides an "executor image", `gcr.io/kaniko-project/executor`

- When running that image, we need to specify at least:

  - the path to the build context (=the directory with our Dockerfile)

  - the target image name (including the registry address)

- Simplified example:

```
docker run \
  -v ...:/workspace gcr.io/kaniko-project/executor \
  --context=/workspace \
  --destination=registry:5000/image_name:image_tag
```

Running Kaniko in a Docker container

- Let's build the image for the DockerCoins `worker` service with Kaniko

Exercise

- Find the port number for our self-hosted registry:

```
kubectl get svc registry
PORT=$(kubectl get svc registry -o json | jq .spec.ports[0].nodePort)
```

- Run Kaniko:

```
docker run --net host \
  -v ~/container.training/dockercoins/worker:/workspace \
  gcr.io/kaniko-project/executor \
  --context=/workspace \
  --destination=127.0.0.1:$PORT/worker-kaniko:latest
```

We use `--net host` so that we can connect to the registry over `127.0.0.1`.

Running Kaniko in a Kubernetes pod

- We need to mount or copy the build context to the pod
- We are going to build straight from the git repository

  (to avoid depending on files sitting on a node, outside of containers)
- We need to `git clone` the repository before running Kaniko
- We are going to use two containers sharing a volume:
  - a first container to `git clone` the repository to the volume
  - a second container to run Kaniko, using the content of the volume
- However, we need the first container to be done before running the second one

🤔 How could we do that?

Init Containers to the rescue

- A pod can have a list of `initContainers`

- `initContainers` are executed in the specified order

- Each Init Container needs to complete (exit) successfully

- If any Init Container fails (non-zero exit status) the pod fails

  (what happens next depends on the pod's `restartPolicy`)

- After all Init Containers have run successfully, normal `containers` are started

- We are going to execute the `git clone` operation in an Init Container

Our Kaniko builder pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kaniko-build
spec:
  initContainers:
  - name: git-clone
    image: alpine
    command: ["sh", "-c"]
    args:
    - |
      apk add --no-cache git &&
      git clone git://github.com/jpetazzo/container.training /workspace
    volumeMounts:
    - name: workspace
      mountPath: /workspace
  containers:
  - name: build-image
    image: gcr.io/kaniko-project/executor:latest
    args:
      - "--context=/workspace/dockercoins/rng"
      - "--insecure"
      - "--destination=registry:5000/rng-kaniko:latest"
    volumeMounts:
    - name: workspace
      mountPath: /workspace
  volumes:
  - name: workspace
```

Explanations

- We define a volume named `workspace` (using the default `emptyDir` provider)
- That volume is mounted to `/workspace` in both our containers
- The `git-clone` Init Container installs `git` and runs `git clone`
- The `build-image` container executes Kaniko
- We use our self-hosted registry DNS name (`registry`)
- We add `--insecure` to use plain HTTP to talk to the registry

Running our Kaniko builder pod

- The YAML for the pod is in `k8s/kaniko-build.yaml`

Exercise

- Create the pod:

```
kubectl apply -f ~/container.training/k8s/kaniko-build.yaml
```

- Watch the logs:

```
stern kaniko
```

Discussion

What should we use? The Docker build technique shown earlier? Kaniko? Something else?

- The Docker build technique is simple, and has the potential to be very fast
- However, it doesn't play nice with Kubernetes resource limits
- Kaniko plays nice with resource limits
- However, it's slower (there is no caching at all)
- The ultimate building tool will probably be Jessica Frazelle's img builder

  (it depends on upstream changes that are not in Kubernetes 1.11.2 yet)

But … is it all about speed? (No!)

The big picture

- For starters: the Docker Hub automated builds are very easy to set up
  - link a GitHub repository with the Docker Hub
  - each time you push to GitHub, an image gets build on the Docker Hub
- If this doesn't work for you: why?
  - too slow (I'm far from `us-east-1`!) → consider using your cloud provider's registry
  - I'm not using a cloud provider → ok, perhaps you need to self-host then
  - I need fancy features (e.g. CI) → consider something like GitLab

Managing configuration

Managing configuration

- Some applications need to be configured (obviously!)
- There are many ways for our code to pick up configuration:
  - command-line arguments
  - environment variables
  - configuration files
  - configuration servers (getting configuration from a database, an API...)
  - ... and more (because programmers can be very creative!)
- How can we do these things with containers and Kubernetes?

Passing configuration to containers

- There are many ways to pass configuration to code running in a container:
  - baking it in a custom image
  - command-line arguments
  - environment variables
  - injecting configuration files
  - exposing it over the Kubernetes API
  - configuration servers
- Let's review these different strategies!

Baking custom images

- Put the configuration in the image

  (it can be in a configuration file, but also `ENV` or `CMD` actions)

- It's easy! It's simple!

- Unfortunately, it also has downsides:

  - multiplication of images

  - different images for dev, staging, prod ...

  - minor reconfigurations require a whole build/push/pull cycle

- Avoid doing it unless you don't have the time to figure out other options

Command-line arguments

- Pass options to `args` array in the container specification

- Example (source):

```yaml
args:
  - "--data-dir=/var/lib/etcd"
  - "--advertise-client-urls=http://127.0.0.1:2379"
  - "--listen-client-urls=http://127.0.0.1:2379"
  - "--listen-peer-urls=http://127.0.0.1:2380"
  - "--name=etcd"
```

- The options can be passed directly to the program that we run ...

  ... or to a wrapper script that will use them to e.g. generate a config file

Command-line arguments, pros & cons

- Works great when options are passed directly to the running program

  (otherwise, a wrapper script can work around the issue)

- Works great when there aren't too many parameters

  (to avoid a 20-lines `args` array)

- Requires documentation and/or understanding of the underlying program

  ("which parameters and flags do I need, again?")

- Well-suited for mandatory parameters (without default values)

- Not ideal when we need to pass a real configuration file anyway

Environment variables

- Pass options through the `env` map in the container specification
- Example:

```yaml
env:
- name: ADMIN_PORT
  value: "8080"
- name: ADMIN_AUTH
  value: Basic
- name: ADMIN_CRED
  value: "admin:0pensesame!"
```

⚠️ `value` must be a string! Make sure that numbers and fancy strings are quoted.

🤔 Why this weird `{name: xxx, value: yyy}` scheme? It will be revealed soon!

The downward API

- In the previous example, environment variables have fixed values
- We can also use a mechanism called the *downward API*
- The downward API allows to expose pod or container information
  - either through special files (we won't show that for now)
  - or through environment variables
- The value of these environment variables is computed when the container is started
- Remember: environment variables won't (can't) change after container start
- Let's see a few concrete examples!

Exposing the pod's namespace

```
  - name: MY_POD_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

- Useful to generate FQDN of services

  (in some contexts, a short name is not enough)

- For instance, the two commands should be equivalent:

  ```
  curl api-backend
  curl api-backend.$MY_POD_NAMESPACE.svc.cluster.local
  ```

Exposing the pod's IP address

```
    - name: MY_POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
```

- Useful if we need to know our IP address

  (we could also read it from `eth0`, but this is more solid)

Exposing the container's resource limits

```
- name: MY_MEM_LIMIT
  valueFrom:
    resourceFieldRef:
      containerName: test-container
      resource: limits.memory
```

- Useful for runtimes where memory is garbage collected

- Example: the JVM

  (the memory available to the JVM should be set with the `-Xmx` flag)

- Best practice: set a memory limit, and pass it to the runtime

  (see this blog post for a detailed example)

More about the downward API

- This documentation page tells more about these environment variables
- And this one explains the other way to use the downward API

  (through files that get created in the container filesystem)

Environment variables, pros and cons

- Works great when the running program expects these variables
- Works great for optional parameters with reasonable defaults

  (since the container image can provide these defaults)

- Sort of auto-documented

  (we can see which environment variables are defined in the image, and their values)

- Can be (ab)used with longer values ...
- ... You *can* put an entire Tomcat configuration file in an environment ...
- ... But *should* you?

(Do it if you really need to, we're not judging! But we'll see better ways.)

Injecting configuration files

- Sometimes, there is no way around it: we need to inject a full config file
- Kubernetes provides a mechanism for that purpose: `configmaps`
- A configmap is a Kubernetes resource that exists in a namespace
- Conceptually, it's a key/value map

  (values are arbitrary strings)
- We can think about them in (at least) two different ways:
  - as holding entire configuration file(s)
  - as holding individual configuration parameters

Note: to hold sensitive information, we can use "Secrets", which are another type of resource behaving very much like configmaps. We'll cover them just after!

Configmaps storing entire files

- In this case, each key/value pair corresponds to a configuration file

- Key = name of the file

- Value = content of the file

- There can be one key/value pair, or as many as necessary

  (for complex apps with multiple configuration files)

- Examples:

```
# Create a configmap with a single key, "app.conf"
kubectl create configmap my-app-config --from-file=app.conf
# Create a configmap with a single key, "app.conf" but another file
kubectl create configmap my-app-config --from-file=app.conf=app-prod.conf
# Create a configmap with multiple keys (one per file in the config.d directory)
kubectl create configmap my-app-config --from-file=config.d/
```

Configmaps storing individual parameters

- In this case, each key/value pair corresponds to a parameter
- Key = name of the parameter
- Value = value of the parameter
- Examples:

```
# Create a configmap with two keys
kubectl create cm my-app-config \
  --from-literal=foreground=red \
  --from-literal=background=blue

# Create a configmap from a file containing key=val pairs
kubectl create cm my-app-config \
  --from-env-file=app.conf
```

Exposing configmaps to containers

- Configmaps can be exposed as plain files in the filesystem of a container
  - this is achieved by declaring a volume and mounting it in the container
  - this is particularly effective for configmaps containing whole files
- Configmaps can be exposed as environment variables in the container
  - this is achieved with the downward API
  - this is particularly effective for configmaps containing individual parameters
- Let's see how to do both!

Passing a configuration file with a configmap

- We will start a load balancer powered by HAProxy
- We will use the official `haproxy` image
- It expects to find its configuration in `/usr/local/etc/haproxy/haproxy.cfg`
- We will provide a simple HAproxy configuration, `k8s/haproxy.cfg`
- It listens on port 80, and load balances connections between IBM and Google

Creating the configmap

Exercise

- Go to the `k8s` directory in the repository:

```
cd ~/container.training/k8s
```

- Create a configmap named `haproxy` and holding the configuration file:

```
kubectl create configmap haproxy --from-file=haproxy.cfg
```

- Check what our configmap looks like:

```
kubectl get configmap haproxy -o yaml
```

Using the configmap

We are going to use the following pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: haproxy
spec:
  volumes:
  - name: config
    configMap:
      name: haproxy
  containers:
  - name: haproxy
    image: haproxy
    volumeMounts:
    - name: config
      mountPath: /usr/local/etc/haproxy/
```

Using the configmap

- The resource definition from the previous slide is in `k8s/haproxy.yaml`

Exercise

- Create the HAProxy pod:

```
kubectl apply -f ~/container.training/k8s/haproxy.yaml
```

- Check the IP address allocated to the pod:

```
kubectl get pod haproxy -o wide
IP=$(kubectl get pod haproxy -o json | jq -r .status.podIP)
```

Testing our load balancer

- The load balancer will send:
  - half of the connections to Google
  - the other half to IBM

Exercise

- Access the load balancer a few times:

```
curl $IP
curl $IP
curl $IP
```

We should see connections served by Google, and others served by IBM.
(Each server sends us a redirect page. Look at the URL that they send us to!)

Exposing configmaps with the downward API

- We are going to run a Docker registry on a custom port
- By default, the registry listens on port 5000
- This can be changed by setting environment variable `REGISTRY_HTTP_ADDR`
- We are going to store the port number in a configmap
- Then we will expose that configmap to a container environment variable

Creating the configmap

Exercise

- Our configmap will have a single key, `http.addr`:

```
kubectl create configmap registry --from-literal=http.addr=0.0.0.0:80
```

- Check our configmap:

```
kubectl get configmap registry -o yaml
```

Using the configmap

We are going to use the following pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: registry
spec:
  containers:
  - name: registry
    image: registry
    env:
    - name: REGISTRY_HTTP_ADDR
      valueFrom:
        configMapKeyRef:
          name: registry
          key: http.addr
```

Using the configmap

- The resource definition from the previous slide is in `k8s/registry.yaml`

Exercise

- Create the registry pod:

```
kubectl apply -f ~/container.training/k8s/registry.yaml
```

- Check the IP address allocated to the pod:

```
kubectl get pod registry -o wide
IP=$(kubectl get pod registry -o json | jq -r .status.podIP)
```

- Confirm that the registry is available on port 80:

```
curl $IP/v2/_catalog
```

Passwords, tokens, sensitive information

- For sensitive information, there is another special resource: *Secrets*

- Secrets and Configmaps work almost the same way

  (we'll expose the differences on the next slide)

- The *intent* is different, though:

  "You should use secrets for things which are actually secret like API keys, credentials, etc., and use config map for not-secret configuration data."

  "In the future there will likely be some differentiators for secrets like rotation or support for backing the secret API w/ HSMs, etc."

  (Source: the author of both features)

Differences between configmaps and secrets

- Secrets are base64-encoded when shown with `kubectl get secrets -o yaml`
  - keep in mind that this is just *encoding*, not *encryption*
  - it is very easy to automatically extract and decode secrets
- Secrets can be encrypted at rest
- With RBAC, we can authorize a user to access configmaps, but not secrets

  (since they are two different kinds of resources)

Owners and dependents

Previous section | Back to table of contents | Next section

Owners and dependents

- Some objects are created by other objects

  (example: pods created by replica sets, themselves created by deployments)

- When an *owner* object is deleted, its *dependents* are deleted

  (this is the default behavior; it can be changed)

- We can delete a dependent directly if we want

  (but generally, the owner will recreate another right away)

- An object can have multiple owners

Finding out the owners of an object

- The owners are recorded in the field `ownerReferences` in the `metadata` block

Exercise

- Let's start a replicated `nginx` deployment:

```
kubectl run yanginx --image=nginx --replicas=3
```

- Once it's up, check the corresponding pods:

```
kubectl get pods -l run=yanginx -o yaml | head -n 25
```

These pods are owned by a ReplicaSet named yanginx-xxxxxxxxxx.

Listing objects with their owners

- This is a good opportunity to try the `custom-columns` output!

Exercise

- Show all pods with their owners:

```
kubectl get pod -o custom-columns=\
NAME:.metadata.name,\
OWNER-KIND:.metadata.ownerReferences[0].kind,\
OWNER-NAME:.metadata.ownerReferences[0].name
```

Note: the `custom-columns` option should be one long option (without spaces), so the lines should not be indented (otherwise the indentation will insert spaces).

Deletion policy

- When deleting an object through the API, three policies are available:
  - foreground (API call returns after all dependents are deleted)
  - background (API call returns immediately; dependents are scheduled for deletion)
  - orphan (the dependents are not deleted)
- When deleting an object with `kubectl`, this is selected with `--cascade`:
  - `--cascade=true` deletes all dependent objects (default)
  - `--cascade=false` orphans dependent objects

What happens when an object is deleted

- It is removed from the list of owners of its dependents
- If, for one of these dependents, the list of owners becomes empty ...
  - if the policy is "orphan", the object stays
  - otherwise, the object is deleted

Orphaning pods

- We are going to delete the Deployment and Replica Set that we created
- ... without deleting the corresponding pods!

Exercise

- Delete the Deployment:

```
kubectl delete deployment -l run=yanginx --cascade=false
```

- Delete the Replica Set:

```
kubectl delete replicaset -l run=yanginx --cascade=false
```

- Check that the pods are still here:

```
kubectl get pods
```

When and why would we have orphans?

- If we remove an owner and explicitly instruct the API to orphan dependents

  (like on the previous slide)

- If we change the labels on a dependent, so that it's not selected anymore

  (e.g. change the `run: yanginx` in the pods of the previous example)

- If a deployment tool that we're using does these things for us

- If there is a serious problem within API machinery or other components

  (i.e. "this should not happen")

Finding orphan objects

- We're going to output all pods in JSON format
- Then we will use `jq` to keep only the ones *without* an owner
- And we will display their name

Exercise

- List all pods that *do not* have an owner:

```
kubectl get pod -o json | jq -r "
      .items[]
      | select(.metadata.ownerReferences|not)
      | .metadata.name"
```

Deleting orphan pods

- Now that we can list orphan pods, deleting them is easy

Exercise

- Add `| xargs kubectl delete pod` to the previous command:

```
kubectl get pod -o json | jq -r "
      .items[]
      | select(.metadata.ownerReferences|not)
      | .metadata.name" | xargs kubectl delete pod
```

As always, the documentation has useful extra information and pointers.

Stateful sets

Stateful sets

- Stateful sets are a type of resource in the Kubernetes API

  (like pods, deployments, services...)

- They offer mechanisms to deploy scaled stateful applications

- At a first glance, they look like *deployments*:

  - a stateful set defines a pod spec and a number of replicas *R*

  - it will make sure that *R* copies of the pod are running

  - that number can be changed while the stateful set is running

  - updating the pod spec will cause a rolling update to happen

- But they also have some significant differences

Stateful sets unique features

- Pods in a stateful set are numbered (from 0 to *R-1*) and ordered
- They are started and updated in order (from 0 to *R-1*)
- A pod is started (or updated) only when the previous one is ready
- They are stopped in reverse order (from *R-1* to 0)
- Each pod know its identity (i.e. which number it is in the set)
- Each pod can discover the IP address of the others easily
- The pods can have persistent volumes attached to them

🤔 Wait a minute ... Can't we already attach volumes to pods and deployments?

Volumes and Persistent Volumes

- Volumes are used for many purposes:
  - sharing data between containers in a pod
  - exposing configuration information and secrets to containers
  - accessing storage systems
- The last type of volumes is known as a "Persistent Volume"

Persistent Volumes types

- There are many types of Persistent Volumes available:
  - public cloud storage (GCEPersistentDisk, AWSElasticBlockStore, AzureDisk...)
  - private cloud storage (Cinder, VsphereVolume...)
  - traditional storage systems (NFS, iSCSI, FC...)
  - distributed storage (Ceph, Glusterfs, Portworx...)
- Using a persistent volume requires:
  - creating the volume out-of-band (outside of the Kubernetes API)
  - referencing the volume in the pod description, with all its parameters

Using a Persistent Volume

Here is a pod definition using an AWS EBS volume (that has to be created first):

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-my-ebs-volume
spec:
  containers:
  - image: ...
    name: container-using-my-ebs-volume
    volumeMounts:
    - mountPath: /my-ebs
      name: my-ebs-volume
  volumes:
  - name: my-ebs-volume
    awsElasticBlockStore:
      volumeID: vol-049df61146c4d7901
      fsType: ext4
```

Shortcomings of Persistent Volumes

- Their lifecycle (creation, deletion...) is managed outside of the Kubernetes API

  (we can't just use `kubectl apply/create/delete/...` to manage them)

- If a Deployment uses a volume, all replicas end up using the same volume

- That volume must then support concurrent access

  - some volumes do (e.g. NFS servers support multiple read/write access)

  - some volumes support concurrent reads

  - some volumes support concurrent access for colocated pods

- What we really need is a way for each replica to have its own volume

Persistent Volume Claims

- To abstract the different types of storage, a pod can use a special volume type
- This type is a *Persistent Volume Claim*
- Using a Persistent Volume Claim is a two-step process:
  - creating the claim
  - using the claim in a pod (as if it were any other kind of volume)
- Between these two steps, something will happen behind the scenes:
  - Kubernetes will associate an existing volume with the claim
  - ... or dynamically create a volume if possible and necessary

What's in a Persistent Volume Claim?

- At the very least, the claim should indicate:
  - the size of the volume (e.g. "5 GiB")
  - the access mode (e.g. "read-write by a single pod")
- It can also give extra details, like:
  - which storage system to use (e.g. Portworx, EBS...)
  - extra parameters for that storage system

  e.g.: "replicate the data 3 times, and use SSD media"
- The extra details are provided by specifying a Storage Class

What's a Storage Class?

- A Storage Class is yet another Kubernetes API resource

  (visible with e.g. `kubectl get storageclass` or `kubectl get sc`)

- It indicates which *provisioner* to use

- And arbitrary parameters for that provisioner

  (replication levels, type of disk ... anything relevant!)

- It is necessary to define a Storage Class to use dynamic provisioning

- Conversely, it is not necessary to define one if you will create volumes manually

  (we will see dynamic provisioning in action later)

Defining a Persistent Volume Claim

Here is a minimal PVC:

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
   name: my-claim
spec:
   accessModes:
     - ReadWriteOnce
   resources:
     requests:
       storage: 1Gi
```

Using a Persistent Volume Claim

Here is the same definition as earlier, but using a PVC:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-using-a-claim
spec:
  containers:
  - image: ...
    name: container-using-a-claim
    volumeMounts:
    - mountPath: /my-ebs
      name: my-volume
  volumes:
  - name: my-volume
    persistentVolumeClaim:
      claimName: my-claim
```

Persistent Volume Claims and Stateful sets

- The pods in a stateful set can define a `volumeClaimTemplate`
- A `volumeClaimTemplate` will dynamically create one Persistent Volume Claim per pod
- Each pod will therefore have its own volume
- These volumes are numbered (like the pods)
- When updating the stateful set (e.g. image upgrade), each pod keeps its volume
- When pods get rescheduled (e.g. node failure), they keep their volume

  (this requires a storage system that is not node-local)
- These volumes are not automatically deleted

  (when the stateful set is scaled down or deleted)

Stateful set recap

- A Stateful sets manages a number of identical pods

  (like a Deployment)

- These pods are numbered, and started/upgraded/stopped in a specific order

- These pods are aware of their number

  (e.g., #0 can decide to be the primary, and #1 can be secondary)

- These pods can find the IP addresses of the other pods in the set

  (through a *headless service*)

- These pods can each have their own persistent storage

  (Deployments cannot do that)

Stateful sets in action

- We are going to deploy a Consul cluster with 3 nodes
- Consul is a highly-available key/value store

  (like etcd or Zookeeper)
- One easy way to bootstrap a cluster is to tell each node:
  - the addresses of other nodes
  - how many nodes are expected (to know when quorum is reached)

Bootstrapping a Consul cluster

After reading the Consul documentation carefully (and/or asking around), we figure out the minimal command-line to run our Consul cluster.

```
consul agent -data=dir=/consul/data -client=0.0.0.0 -server -ui \
       -bootstrap-expect=3 \
       -retry-join=X.X.X.X \
       -retry-join=Y.Y.Y.Y
```

- We need to replace X.X.X.X and Y.Y.Y.Y with the addresses of other nodes
- We can specify DNS names, but then they have to be FQDN
- It's OK for a pod to include itself in the list as well
- We can therefore use the same command-line on all nodes (easier!)

Discovering the addresses of other pods

- When a service is created for a stateful set, individual DNS entries are created
- These entries are constructed like this:

  `<name-of-stateful-set>-<n>.<name-of-service>.<namespace>.svc.cluster.local`

- `<n>` is the number of the pod in the set (starting at zero)
- If we deploy Consul in the default namespace, the names could be:
  - `consul-0.consul.default.svc.cluster.local`
  - `consul-1.consul.default.svc.cluster.local`
  - `consul-2.consul.default.svc.cluster.local`

Putting it all together

- The file `k8s/consul.yaml` defines a service and a stateful set

- It has a few extra touches:

  - the name of the namespace is injected through an environment variable

  - a `podAntiAffinity` prevents two pods from running on the same node

  - a `preStop` hook makes the pod leave the cluster when shutdown gracefully

This was inspired by this excellent tutorial by Kelsey Hightower. Some features from the original tutorial (TLS authentication between nodes and encryption of gossip traffic) were removed for simplicity.

Running our Consul cluster

- We'll use the provided YAML file

Exercise

- Create the stateful set and associated service:

```
kubectl apply -f ~/container.training/k8s/consul.yaml
```

- Check the logs as the pods come up one after another:

```
stern consul
```

- Check the health of the cluster:

```
kubectl exec consul-0 consul members
```

Caveats

- We haven't used a `volumeClaimTemplate` here

- That's because we don't have a storage provider yet

  (except if you're running this on your own and your cluster has one)

- What happens if we lose a pod?

  - a new pod gets rescheduled (with an empty state)

  - the new pod tries to connect to the two others

  - it will be accepted (after 1-2 minutes of instability)

  - and it will retrieve the data from the other pods

Failure modes

- What happens if we lose two pods?
  - manual repair will be required
  - we will need to instruct the remaining one to act solo
  - then rejoin new pods
- What happens if we lose three pods? (aka all of them)
  - we lose all the data (ouch)
- If we run Consul without persistent storage, backups are a good idea!

Highly available Persistent Volumes

Previous section | Back to table of contents | Next section

Highly available Persistent Volumes

- How can we achieve true durability?

- How can we store data that would survive the loss of a node?

Highly available Persistent Volumes

- How can we achieve true durability?
- How can we store data that would survive the loss of a node?
- We need to use Persistent Volumes backed by highly available storage systems
- There are many ways to achieve that:
  - leveraging our cloud's storage APIs
  - using NAS/SAN systems or file servers
  - distributed storage systems

Highly available Persistent Volumes

- How can we achieve true durability?
- How can we store data that would survive the loss of a node?
- We need to use Persistent Volumes backed by highly available storage systems
- There are many ways to achieve that:
  - leveraging our cloud's storage APIs
  - using NAS/SAN systems or file servers
  - distributed storage systems
- We are going to see one distributed storage system in action

Our test scenario

- We will set up a distributed storage system on our cluster
- We will use it to deploy a SQL database (PostgreSQL)
- We will insert some test data in the database
- We will disrupt the node running the database
- We will see how it recovers

Portworx

- Portworx is a *commercial* persistent storage solution for containers
- It works with Kubernetes, but also Mesos, Swarm ...
- It provides hyper-converged storage

  (=storage is provided by regular compute nodes)
- We're going to use it here because it can be deployed on any Kubernetes cluster

  (it doesn't require any particular infrastructure)
- We don't endorse or support Portworx in any particular way

  (but we appreciate that it's super easy to install!)

A useful reminder

- We're installing Portworx because we need a storage system

- If you are using AKS, EKS, GKE ... you already have a storage system

  (but you might want another one, e.g. to leverage local storage)

- If you have setup Kubernetes yourself, there are other solutions available too

  - on premises, you can use a good old SAN/NAS

  - on a private cloud like OpenStack, you can use e.g. Cinder

  - everywhere, you can use other systems, e.g. Gluster, StorageOS

Portworx requirements

- Kubernetes cluster ✔️
- Optional key/value store (etcd or Consul) ❌
- At least one available block device ❌

The key-value store

- In the current version of Portworx (1.4) it is recommended to use etcd or Consul
- But Portworx also has beta support for an embedded key/value store
- For simplicity, we are going to use the latter option

  (but if we have deployed Consul or etcd, we can use that, too)

One available block device

- Block device = disk or partition on a disk
- We can see block devices with `lsblk`

  (or `cat /proc/partitions` if we're old school like that!)
- If we don't have a spare disk or partition, we can use a *loop device*
- A loop device is a block device actually backed by a file
- These are frequently used to mount ISO (CD/DVD) images or VM disk images

Setting up a loop device

- We are going to create a 10 GB (empty) file on each node
- Then make a loop device from it, to be used by Portworx

Exercise

- Create a 10 GB file on each node:

```
for N in $(seq 1 4); do ssh node$N sudo truncate --size 10G /portworx.blk; done
```

(If SSH asks to confirm host keys, enter `yes` each time.)

- Associate the file to a loop device on each node:

```
for N in $(seq 1 4); do ssh node$N sudo losetup /dev/loop4 /portworx.blk; done
```

Installing Portworx

- To install Portworx, we need to go to https://install.portworx.com/
- This website will ask us a bunch of questoins about our cluster
- Then, it will generate a YAML file that we should apply to our cluster

Installing Portworx

- To install Portworx, we need to go to https://install.portworx.com/
- This website will ask us a bunch of questoins about our cluster
- Then, it will generate a YAML file that we should apply to our cluster
- Or, we can just apply that YAML file directly (it's in `k8s/portworx.yaml`)

Exercise

- Install Portworx:

```
kubectl apply -f ~/container.training/k8s/portworx.yaml
```

Generating a custom YAML file

If you want to generate a YAML file tailored to your own needs, the easiest way is to use https://install.portworx.com/.

FYI, this is how we obtained the YAML file used earlier:

```
KBVER=$(kubectl version -o json | jq -r .serverVersion.gitVersion)
BLKDEV=/dev/loop4
curl https://install.portworx.com/1.4/?kbver=$KBVER&b=true&s=$BLKDEV&c=px-workshop&stork=tru
```

If you want to use an external key/value store, add one of the following:

```
&k=etcd://XXX:2379
&k=consul://XXX:8500
```

… where `XXX` is the name or address of your etcd or Consul server.

Waiting for Portworx to be ready

- The installation process will take a few minutes

Exercise

- Check out the logs:

```
stern -n kube-system portworx
```

- Wait until it gets quiet

  (you should see `portworx service is healthy`, too)

Dynamic provisioning of persistent volumes

- We are going to run PostgreSQL in a Stateful set
- The Stateful set will specify a `volumeClaimTemplate`
- That `volumeClaimTemplate` will create Persistent Volume Claims
- Kubernetes' dynamic provisioning will satisfy these Persistent Volume Claims

  (by creating Persistent Volumes and binding them to the claims)
- The Persistent Volumes are then available for the PostgreSQL pods

Storage Classes

- It's possible that multiple storage systems are available
- Or, that a storage system offers multiple tiers of storage

  (SSD vs. magnetic; mirrored or not; etc.)
- We need to tell Kubernetes *which* system and tier to use
- This is achieved by creating a Storage Class
- A `volumeClaimTemplate` can indicate which Storage Class to use
- It is also possible to mark a Storage Class as "default"

  (it will be used if a `volumeClaimTemplate` doesn't specify one)

Our default Storage Class

This is our Storage Class (in `k8s/storage-class.yaml`):

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: portworx-replicated
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/portworx-volume
parameters:
 repl: "2"
 priority_io: "high"
```

- It says "use Portworx to create volumes"
- It tells Portworx to "keep 2 replicas of these volumes"
- It marks the Storage Class as being the default one

Creating our Storage Class

- Let's apply that YAML file!

Exercise

- Create the Storage Class:

```
kubectl apply -f ~/container.training/k8s/storage-class.yaml
```

- Check that it is now available:

```
kubectl get sc
```

It should show as `portworx-replicated (default)`.

Our Postgres Stateful set

- The next slide shows `k8s/postgres.yaml`
- It defines a Stateful set
- With a `volumeClaimTemplate` requesting a 1 GB volume
- That volume will be mounted to `/var/lib/postgresql/data`
- There is another little detail: we enable the `stork` scheduler
- The `stork` scheduler is optional (it's specific to Portworx)
- It helps the Kubernetes scheduler to colocate the pod with its volume

  (see this blog post for more details about that)

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  selector:
    matchLabels:
      app: postgres
  serviceName: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      schedulerName: stork
      containers:
      - name: postgres
        image: postgres:10.5
        volumeMounts:
        - mountPath: /var/lib/postgresql/data
          name: postgres
  volumeClaimTemplates:
  - metadata:
      name: postgres
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

Creating the Stateful set

- Before applying the YAML, watch what's going on with `kubectl get events -w`

Exercise

- Apply that YAML:

```
kubectl apply -f ~/container.training/k8s/postgres.yaml
```

Testing our PostgreSQL pod

- We will use `kubectl exec` to get a shell in the pod

- Good to know: we need to use the `postgres` user in the pod

Exercise

- Get a shell in the pod, as the `postgres` user:

```
kubectl exec -ti postgres-0 su postgres
```

- Check that default databases have been created correctly:

```
psql -l
```

(This should show us 3 lines: postgres, template0, and template1.)

Inserting data in PostgreSQL

- We will create a database and populate it with `pgbench`

Exercise

- Create a database named `demo`:

```
createdb demo
```

- Populate it with `pgbench`:

```
pgbench -i -s 10 demo
```

- The `-i` flag means "create tables"

- The `-s 10` flag means "create 10 x 100,000 rows"

Checking how much data we have now

- The `pgbench` tool inserts rows in table `pgbench_accounts`

Exercise

- Check that the `demo` base exists:

```
psql -l
```

- Check how many rows we have in `pgbench_accounts`:

```
psql demo -c "select count(*) from pgbench_accounts"
```

(We should see a count of 1,000,000 rows.)

Find out which node is hosting the database

- We can find that information with `kubectl get pods -o wide`

Exercise

- Check the node running the database:

```
kubectl get pod postgres-0 -o wide
```

We are going to disrupt that node.

Find out which node is hosting the database

- We can find that information with `kubectl get pods -o wide`

Exercise

- Check the node running the database:

```
kubectl get pod postgres-0 -o wide
```

We are going to disrupt that node.

By "disrupt" we mean: "disconnect it from the network".

Disconnect the node

- We will use `iptables` to block all traffic exiting the node

  (except SSH traffic, so we can repair the node later if needed)

Exercise

- SSH to the node to disrupt:

```
ssh nodeX
```

- Allow SSH traffic leaving the node, but block all other traffic:

```
sudo iptables -I OUTPUT -p tcp --sport 22 -j ACCEPT
sudo iptables -I OUTPUT 2 -j DROP
```

enix.io

Check that the node is disconnected

⌨ Exercise

- Check that the node can't communicate with other nodes:

```
ping node1
```

- Logout to go back on `node1`

- Watch the events unfolding with `kubectl get events -w` and `kubectl get pods -w`

- It will take some time for Kubernetes to mark the node as unhealthy

- Then it will attempt to reschedule the pod to another node

- In about a minute, our pod should be up and running again

Check that our data is still available

- We are going to reconnect to the (new) pod and check

Exercise

- Get a shell on the pod:

```
kubectl exec -ti postgres-0 su postgres
```

- Check the number of rows in the `pgbench_accounts` table:

```
psql demo -c "select count(*) from pgbench_accounts"
```

Double-check that the pod has really moved

- Just to make sure the system is not bluffing!

Exercise

- Look at which node the pod is now running on

```
kubectl get pod postgres-0 -o wide
```

Re-enable the node

- Let's fix the node that we disconnected from the network

Exercise

- SSH to the node:

```
ssh nodeX
```

- Remove the iptables rule blocking traffic:

```
sudo iptables -D OUTPUT 2
```

A few words about this PostgreSQL setup

- In a real deployment, you would want to set a password

- This can be done by creating a `secret`:

```
kubectl create secret generic postgres \
      --from-literal=password=$(base64 /dev/urandom | head -c16)
```

- And then passing that secret to the container:

```
env:
- name: POSTGRES_PASSWORD
valueFrom:
  secretKeyRef:
    name: postgres
    key: password
```

Troubleshooting Portworx

- If we need to see what's going on with Portworx:

```
PXPOD=$(kubectl -n kube-system get pod -l name=portworx -o json |
        jq -r .items[0].metadata.name)
kubectl -n kube-system exec $PXPOD -- /opt/pwx/bin/pxctl status
```

- We can also connect to Lighthouse (a web UI)
  - check the port with `kubectl -n kube-system get svc px-lighthouse`
  - connect to that port
  - the default login/password is `admin/Password1`
  - then specify `portworx-service` as the endpoint

Removing Portworx

- Portworx provides a storage driver
- It needs to place itself "above" the Kubelet

  (it installs itself straight on the nodes)
- To remove it, we need to do more than just deleting its Kubernetes resources
- It is done by applying a special label:

  ```
  kubectl label nodes --all px/enabled=remove --overwrite
  ```
- Then removing a bunch of local files:

  ```
  sudo chattr -i /etc/pwx/.private.json
  sudo rm -rf /etc/pwx /opt/pwx
  ```

  (on each node where Portworx was running)

Dynamic provisioning without a provider

- What if we want to use Stateful sets without a storage provider?
- We will have to create volumes manually

  (by creating Persistent Volume objects)
- These volumes will be automatically bound with matching Persistent Volume Claims
- We can use local volumes (essentially bind mounts of host directories)
- Of course, these volumes won't be available in case of node failure
- Check this blog post for more information and gotchas

Acknowledgements

The Portworx installation tutorial, and the PostgreSQL example, were inspired by Portworx examples on Katacoda, in particular:

- installing Portworx on Kubernetes

  (with adapatations to use a loop device and an embedded key/value store)

- persistent volumes on Kubernetes using Portworx

  (with adapatations to specify a default Storage Class)

- HA PostgreSQL on Kubernetes with Portworx

  (with adaptations to use a Stateful Set and simplify PostgreSQL's setup)

Next steps

Next steps

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

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!

Options for our first production cluster

- Get a managed cluster from a major cloud provider (AKS, EKS, GKE...)

  (price: $, difficulty: medium)

- Hire someone to deploy it for us

  (price: $$, difficulty: easy)

- Do it ourselves

  (price: $-$$$, difficulty: hard)

One big cluster vs. multiple small ones

- Yes, it is possible to have prod+dev in a single cluster

  (and implement good isolation and security with RBAC, network policies...)

- But it is not a good idea to do that for our first deployment

- Start with a production cluster + at least a test cluster

- Implement and check RBAC and isolation on the test cluster

  (e.g. deploy multiple test versions side-by-side)

- Make sure that all our devs have usable dev clusters

  (whether it's a local minikube or a full-blown multi-node cluster)

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)

Managing stack deployments

- The best deployment tool will vary, depending on:

  - the size and complexity of your stack(s)
  - how often you change it (i.e. add/remove components)
  - the size and skills of your team

- A few examples:

  - shell scripts invoking `kubectl`
  - YAML resources descriptions committed to a repo
  - Helm (~package manager)
  - Spinnaker (Netflix' CD platform)
  - Brigade (event-driven scripting; no YAML)

Cluster federation

Cluster federation

Cluster federation

Sorry Star Trek fans, this is not the federation you're looking for!

Cluster federation

Sorry Star Trek fans, this is not the federation you're looking for!

(If I add "Your cluster is in another federation" I might get a 3rd fandom wincing!)

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?

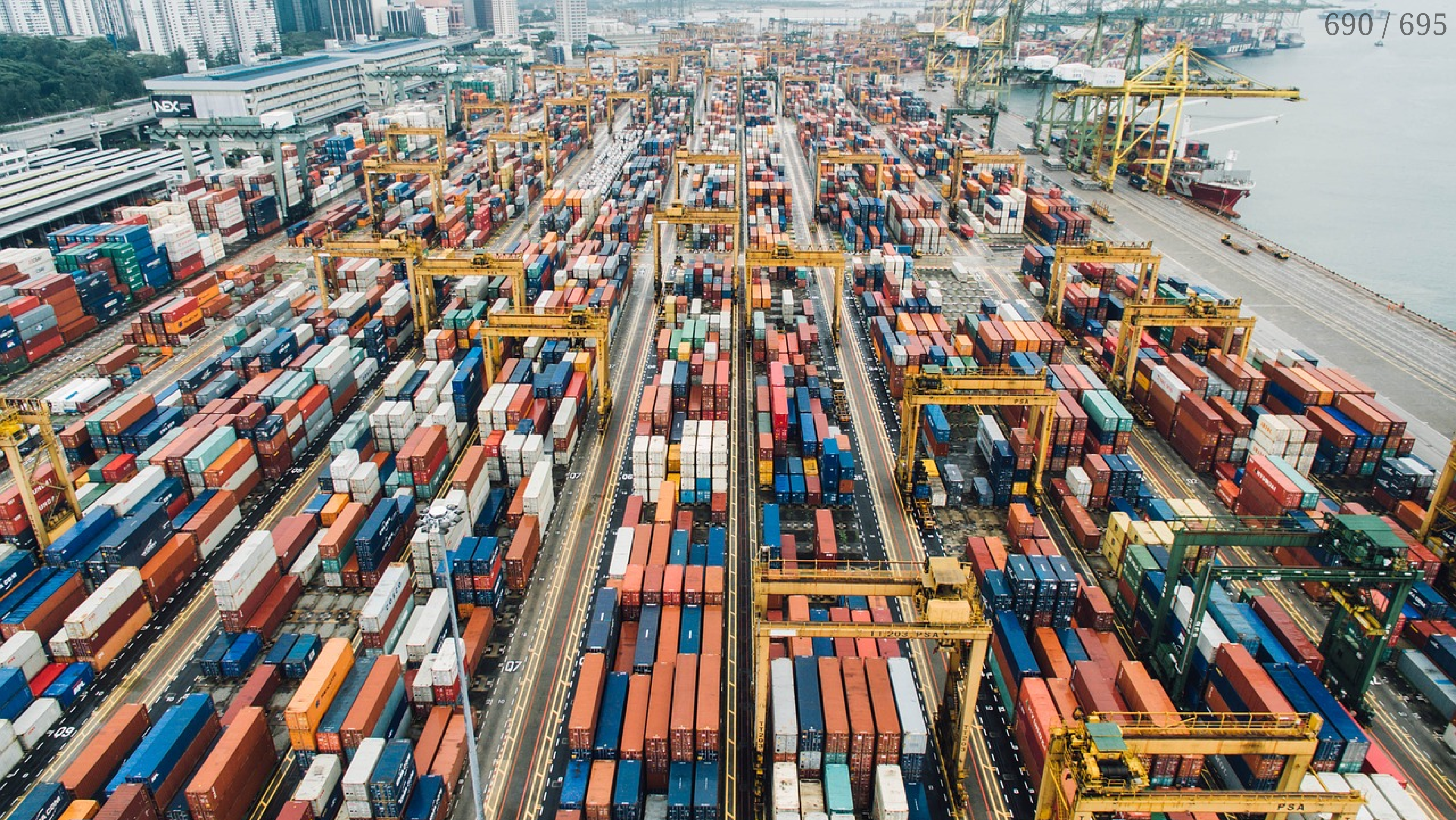
Cluster federation

- 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

Developer experience

We've put this last, but it's pretty important!

- How do you on-board a new developer?
- What do they need to install to get a dev stack?
- How does a code change make it from dev to prod?
- How does someone add a component to a stack?


NEX
PSA
LT203 PSA
07

Links and resources

Links and resources

All things Kubernetes:

- Kubernetes Community - Slack, Google Groups, meetups
- Kubernetes on StackOverflow
- Play With Kubernetes Hands-On Labs

All things Docker:

- Docker documentation
- Docker Hub
- Docker on StackOverflow
- Play With Docker Hands-On Labs

Everything else:

- Local meetups

These slides (and future updates) are on → http://container.training/

Final words

Final words

- For $$$ reasons, our clusters will be shut down *now*
  - if you want another cluster that you can keep longer, come talk to me
- If you liked this tutorial:
  - rate it on the O'Reilly website
  - tweet about it, tagging @jpetazzo and #VelocityConf
  - hire me to deliver it for your team: jerome.petazzoni@gmail.com
- If you didn't like this tutorial:
  - please tell me why, so I can do better next time!