



Deploying Microservices with Kubernetes

Presented by **Travis Holton**

Deploying Microservices with Kubernetes



Course Outline

Setting up

Course Prerequisites

- Account with **Docker Hub**
 - take a few minutes to set one up
- Linux/Unix
 - working with a shell
 - navigating directories
- Basic understanding of Docker
 - commands and usage
 - How to run containers

Training Environment

- In this course we'll be using
 - Docker Community Edition
 - docker - compose
 - minikube
- Training machines should be setup
 - Ubuntu (xenial)
- Following slides contain instructions for setting up environment

Install some dependencies

- Docker Community Edition
- docker-compose
- minikube
 - virtualbox

Courseslides

- Check that your environment contains following directories
 - kubernetes-microservices-workshop
 - example-voting-app
- Run the course slides

```
cd ~/kubernetes-microservices-workshop/slides  
npm start
```

- This should open a browser with the slides for this workshop

Microservices

Some perspective: Monolithic Applications

- Often standard MVC architecture
 - Frontend *view* HTML, JavaScript
 - Backend *controller*
 - Database *model*
- Typically one large binary to deploy
- Single command to run

Disadvantages of Monolithic Apps

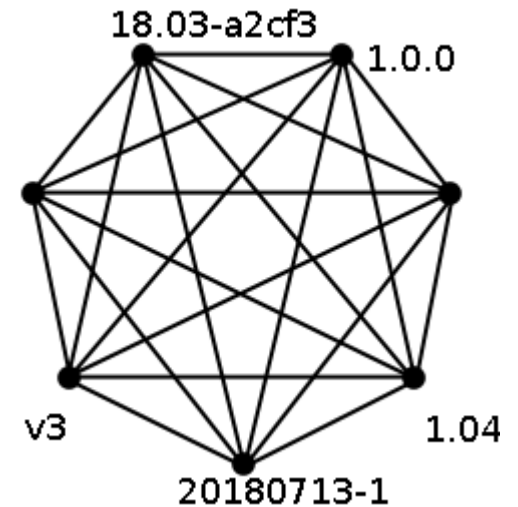
- Tightly coupled code
 - Smallest change still requires complete release
 - Small change can affect entire application
 - Long release cycles
- Usually single large DB
 - Tight coupling with data model
 - Simple schema change can break app
- Matching demand
 - Horizontal scaling
- Rollback of an entire version

Microservices

- Application consists of multiple *components*
 - A *service* is fundamental component
- Services are isolated in scope and functionality
 - Lowers impact of change
- Services can be coupled to separate storage backends
 - Reduce impact of schema changes on entire app
- Can be scaled independently
- Smaller memory/CPU fingerprint means faster deployment

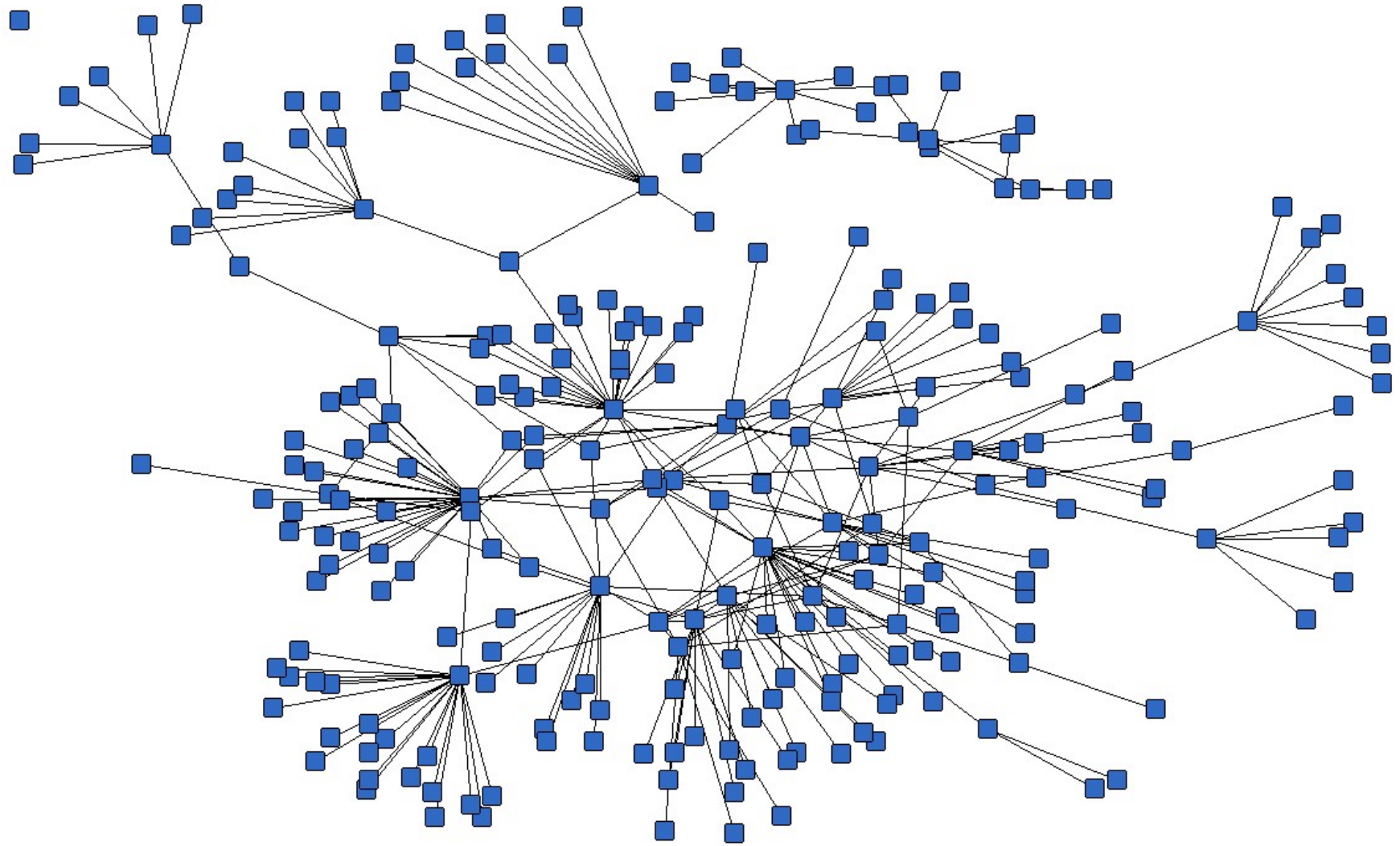
Disadvantages of Microservices

- Remote calls more expensive
- Services must agree on a protocol
 - XML (XMLRPC, SOAP)
 - JSON
- Can become complicated to version individual services



Managing Microservices

- How do you manage 10 or 100s of microservices?



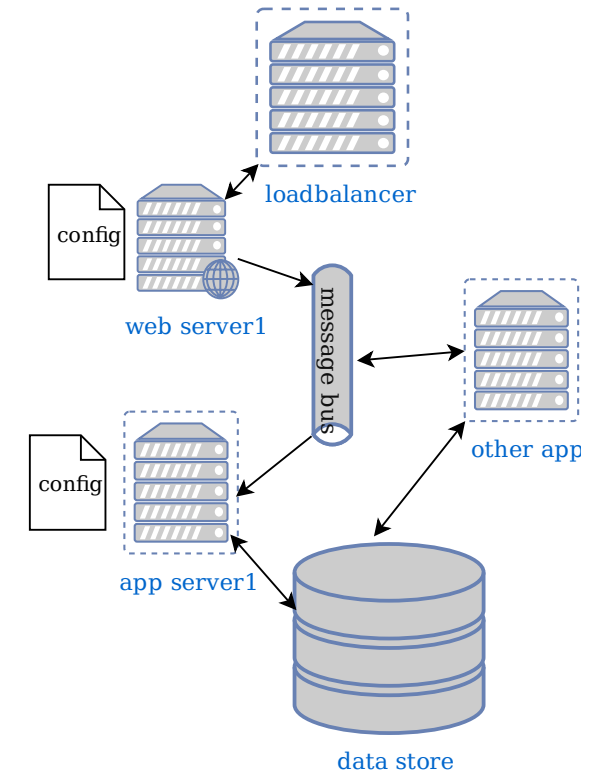
Literature about Microservices

- **Microservices:** Martin Fowler and James Lewis

Developing Microservices

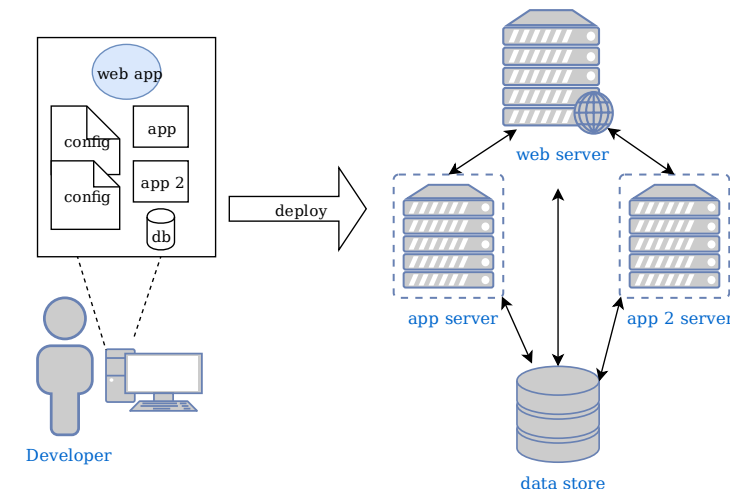
Developing Applications

- Applications can consist of many components
 - Web server (nginx, apache)
 - Database (sql, nosql)
 - Message Queue
 - Your application
- Typically spread across cluster of machines



Single VM Development

- Often desirable for dev to be similar to production environment
- In practice this is often difficult to achieve
 - Limited CPU of dev machines
 - Cost of machines
- Compromise is to develop everything in single VM



Pitfalls of Single VM Development

- Single VM development creates blindspot
- Developers can make false assumptions about
 - Which config files on which machines
 - Which dependency libraries present on machines
- Difficult to scale individual services
- Applications components often tightly coupled
- Can lead to unpredictable behaviour when application is deployed to production

Containers to the rescue

- Containers can make this easier
- Container serves as the *unit* equivalent of microservice
- Deployable artefact (i.e. *image*)
- Can be versioned
- Layered filesystem
 - Deploying updates equivalent of deploying *just what was changed*

Using containers in development

- So how can we run a containerised application in development?
- `docker - compose`
 - A tool for easily bootstrapping multi-container applications
 - i.e. microservice
 - Ideal for developing/testing applications on a workstation
- Let's create an application using `docker - compose`

Exercise: Create microservices docker - compose

```
cd ~/kubernetes-microservices-workshop/sample-code/mycomposeapp
```

- Let's build a simple application with two components
 - Web application using Python Flask
 - Redis message queue
- The app is already in mycomposeapp/app.py
- We want to run the app and redis as separate microservices
- Redis is already available as a **docker image**

Create Our App

- Fire up your favourite editor and create a Dockerfile
 - For example

```
vim Dockerfile
```

- Add the following contents to Dockerfile

```
FROM python:3.7-alpine
WORKDIR /code
COPY requirements.txt /code
RUN pip install -r requirements.txt
COPY . /code
CMD ["python", "app.py"]
```

Writing a docker-compose file

- In the same directory as previous example
- Create a file called `docker-compose.yml`
- Add our service definition:

```
---  
version: "3"  
services:  
  web:  
    build: .  
    ports:  
      - "5000:5000"  
  redis:  
    image: redis:alpine
```

- Start our microservices

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

- Confirm application running at **localhost:5000**

What did docker-compose do?

- Check that services are running

```
docker-compose ps
```

- Have a look at `docker-compose.yml`
- Compose created two services
 - *web* - which automatically built using our Dockerfile
 - *redis* - pulled from the official `redis:alpine`
- Port 5000 on host machine mapped to 5000 on *web* service

Benefits of docker - compose

- Orchestration of containers (a.k.a *services*)
- Mount directories and *named volumes*
 - persistent storage for databases
 - local filesystem for development
- Additional features
 - create multiple networks
 - scale *services*
- *A production* deployment platform?

Scaling Services

- Try scaling the redis service to 4 instances

```
docker-compose up -d --scale redis=4
```


Stopping docker-compose

- In the directory where your `docker-compose.yml` file is, run:

```
docker-compose stop
```

Summary

- docker - compose provides useful way to setup development environments
- Takes care of
 - networking
 - linking containers
 - scaling services

Sample Microservice Application

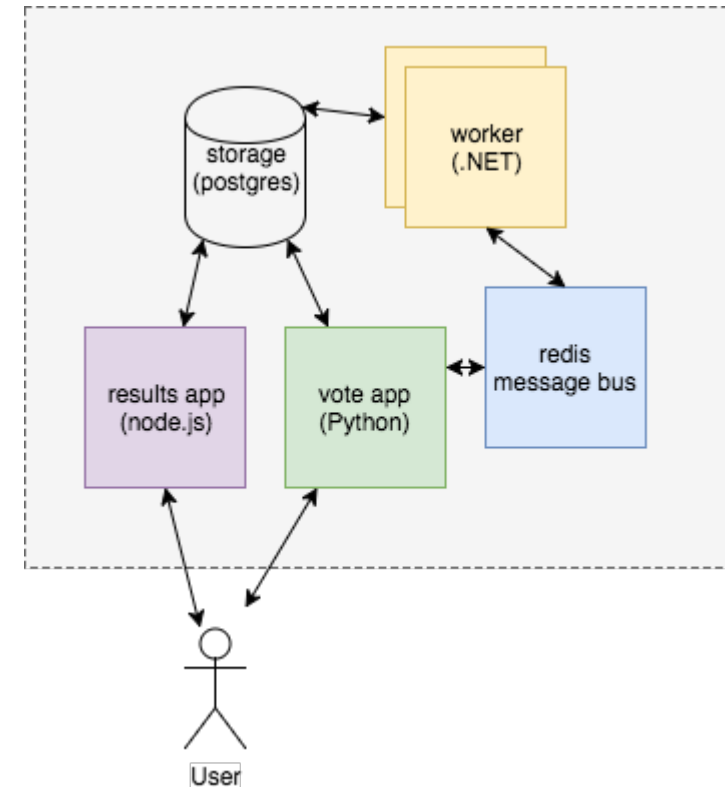
Building a microservice application

- We're going to build the **Example Voting App** tutorial from Docker
- Clone the repository if it's not on your machine already

```
git clone https://github.com/docker/example-voting-app.git
```

The Voting Application

- A polyglot application
- 5 components:
 - Python web application
 - Redis queue
 - .NET worker
 - Postgres DB with a data volume
 - Node.js app to show votes in real time



Start Application

```
cd ~/example-voting-app  
docker-compose up
```

- This may take some time to download/build images
- Once completed you can **vote** and **view results**

Interactive development

- Open up `vote/app.py`

```
vim vote/app.py
```

- On lines 8 & 9, modify vote options
- View change in **voting** application

Change Vote Options

Login to Docker Hub

- In order to push images you'll need to login to docker.io

```
docker login
```

```
Username: YOURNAME
```

```
Password: ****
```

Exercise: Package your update into an image

- Tell docker-compose to rebuild the voting app image

```
docker-compose build vote
```

- This will build `example-voting-app_vote:latest`

- Tag the image

```
docker tag example-voting-app_vote:latest YOURNAME/vote:v2
```

- Push to hub.docker.com

```
docker push YOURNAME/vote:v2
```

Exercise: Change Result Display

- Results are rendered in the NodeJS application
- Edit the view template to replace *Cats* and *Dogs* with your own options

```
vim ~/example-voting-app/result/views/index.html
```

- Repeat previous steps to ship
YOURNAME/result:v2 to DockerHub

Developer Workflow

- Push code to repository
- Continuous Integration (CI) system runs tests
- If tests successful, automate image build & push to a docker registry
- Easy to setup Docker build pipelines with existing services
 - DockerHub
 - GitHub
 - CircleCI
 - Quay.io

Summary

- With docker-compose it's relatively easy to develop on a microservice application
- Changes visible in real time
- Can easily package and distribute images for others to use

Orchestration

Managing Microservice Containers

- Still quite a lot of work maintaining large applications
- Monitoring health of many services
- Maintain homeostasis
 - constant number of *healthy* services
 - scale in proportion to demand
- Next step: orchestration platforms

Goals of Container Orchestration

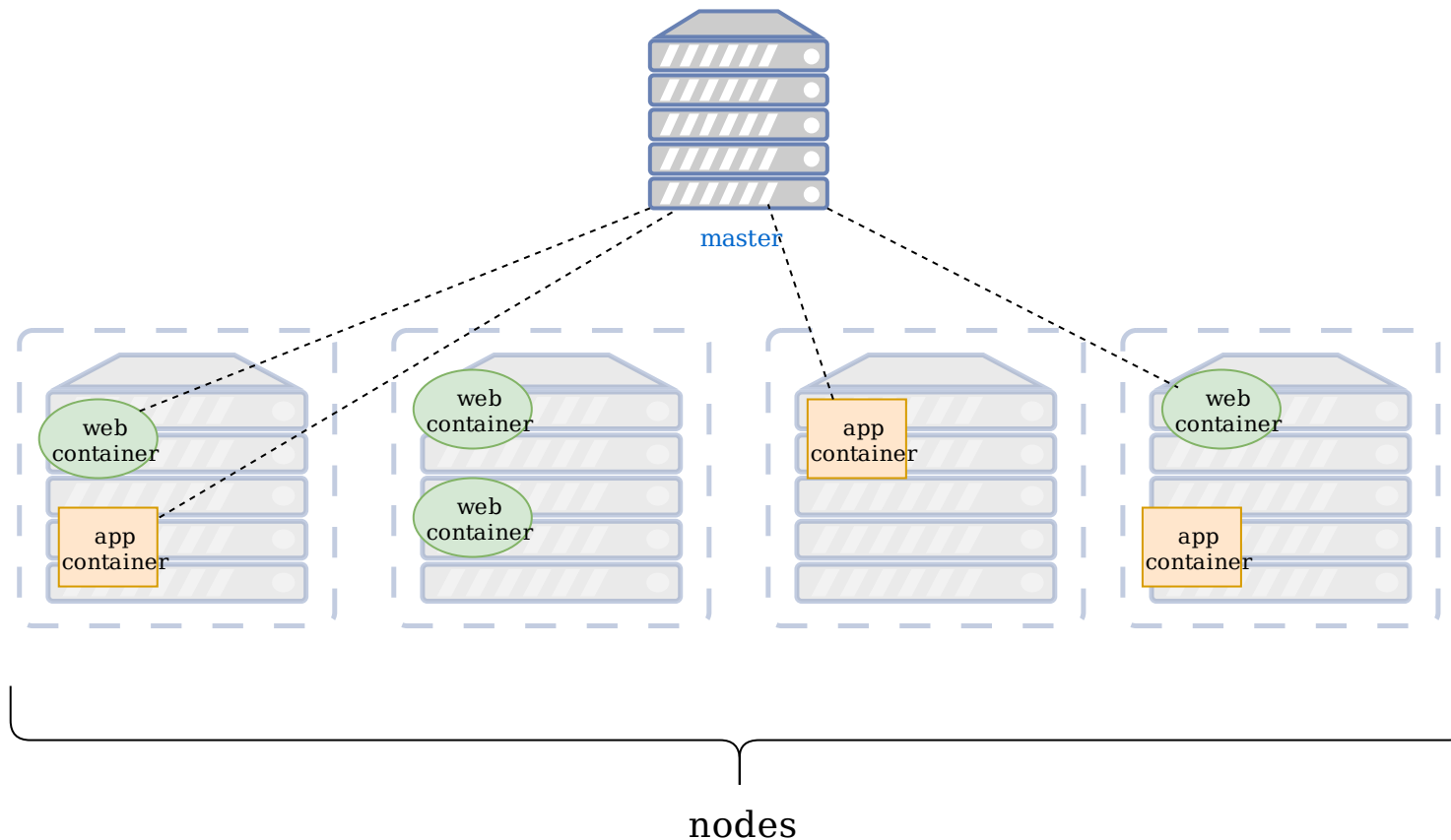
- Monitors health of services
 - Cull unhealthy services
 - Periodically spawn new services
- React to change in demand (autoscaling)
- Manage networking between services

Container Orchestration

- Provides tools for managing containers across a cluster
 - networking
 - scaling
 - monitoring
- Ideal for deploying containerised applications in production

Server Architecture

- Machine designated the *master* or *manager*
- Several machines designated *workers* or *nodes*

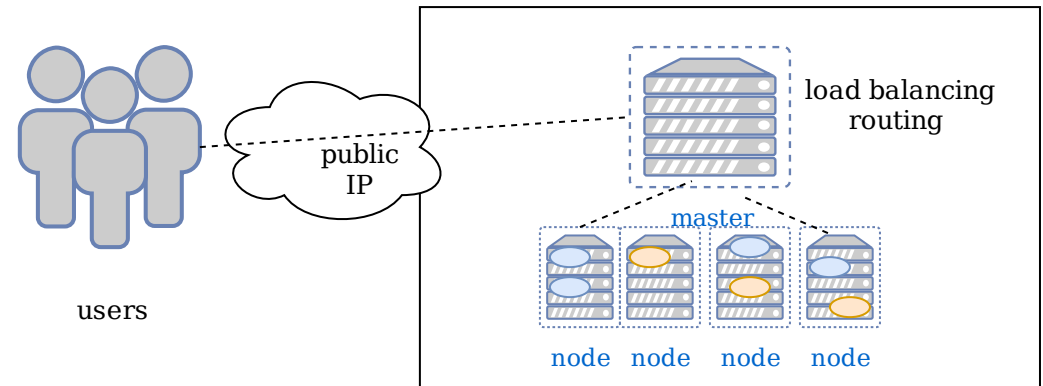


Masters and Nodes

- The *master* is responsible for
 - scheduling containers to run across all *nodes*
 - managing the network interaction between nodes
 - monitoring container health
 - periodically kill/respawn containers
- The *nodes* or *workers*
 - Just run the containers

Container Orchestration: User perspective

- Orchestration framework handles
 - Routing requests to containers
 - Load balancing between different containers
- From user perspective as if interacting with single application



Container Lifecycle

- Containers are ephemeral
- The job of the *master* is to make sure containers are healthy
- It will periodically kill and respawn a container
- This is very similar to *phoenix* principle
 - Outdated or unhealthy containers *burned*
 - Fresh containers spawned in their place



Orchestration Platforms

- Vendor
 - AWS ECS
 - Azure container services
 - Google GKE (based on Kubernetes)
- Docker Swarm
 - Integrated into Docker since mid 2016
- Kubernetes
 - Descends from *Borg*, (Google)
 - Supports multiple container runtimes (eg. Docker, rkt)

Kubernetes

About Kubernetes

- Greek word for *helmsman* or *pilot*
- Also origin of words like *cybernetics* and *government*
- Inspired by *Borg*, Google's internal scheduling tool
- Play on *Borg cube*
- **Source**

What's next

- In the coming slides we will interact with a local Kubernetes instance on training pc
 - minikube
- Along the way we will learn how to interact with and control Kubernetes
- Become acquainted with core Kubernetes concepts
 - Nodes
 - Pods
 - Services
 - etc.

Kubernetes on your workstation



minikube

Minikube

- Creates a *single node* Kubernetes cluster on your PC
- Useful if you
 - want to become familiar with Kubernetes concepts
 - do not have a cluster
 - are preparing/attending a Kubernetes workshop

What you need to run Minikube

- minikube binaries
- A hypervisor
 - KVM
 - virtualbox
- The training machines should be set up

Start Minikube

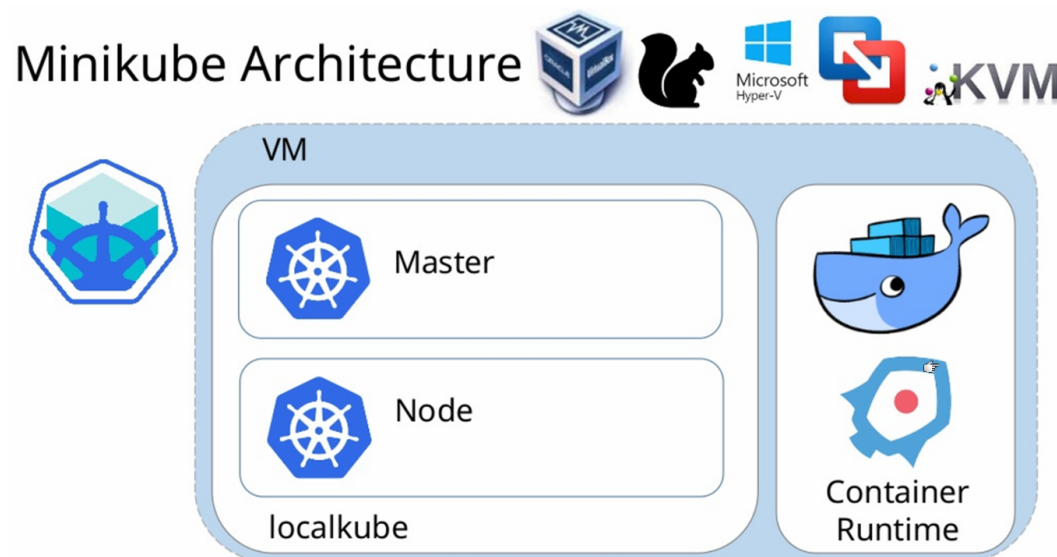
- Start minikube

```
minikube start
```

- May take a few minutes to run while it starts your cluster

Your Minikube Cluster

- You now have a functioning Kubernetes cluster on your workstation
- Single machine playing both roles
 - master
 - node



Interacting with minikube

- General usage instructions

```
minikube help
```

- Open the minikube dashboard

```
minikube dashboard &
```

- will open a tab in your browser

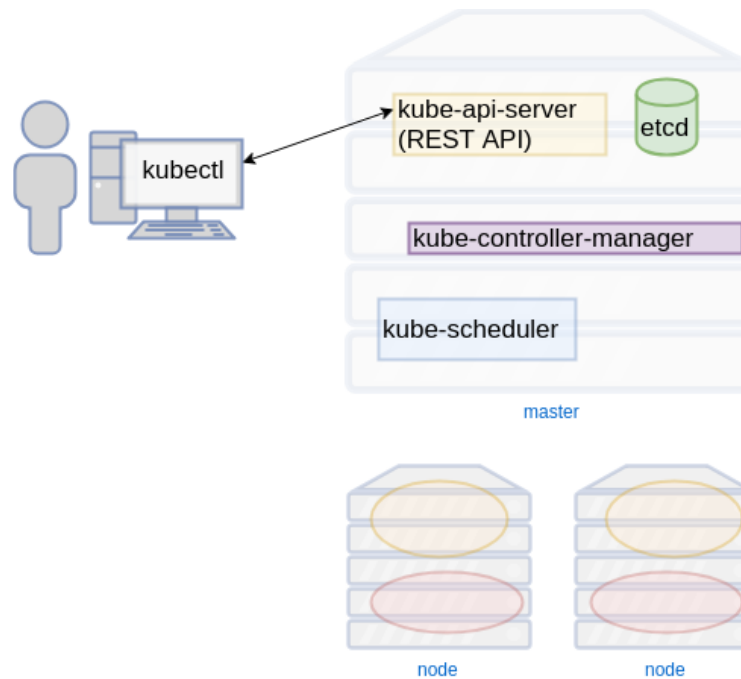
Controlling Kubernetes

kubectl

- The client tool for interacting with Kubernetes REST API
- Tons of functionality
- Pronounced:
 - *cube C T L*
 - *cube C D L*
 - *cube cuddle*

Kubernetes Control Plane

- kubectl
 - client API interface for kubernetes control plane
- api server
 - REST frontend
- controller manager
 - replication
 - deployment
- etcd
 - key/value storage



Inline documentation

kubectl -h

- Use -h option to get an overview of commands

```
$ kubectl -h
kubectl controls the Kubernetes cluster manager.

Find more information at: https://kubernetes.io/docs/reference/kubectl/overview

Basic Commands (Beginner):
  create          Create a resource from a file or from stdin.
  expose          Take a replication controller, service, deployment
```

- Note that they are sorted from *beginner* to *advanced*

Command documentation

kubectl **COMMAND** -h

- Get usage for any commands

```
$ kubectl run -h
Create and run a particular image, possibly replicated.

Creates a deployment or job to manage the created container(s).

Examples:
# Start a single instance of nginx.
kubectl run nginx --image=nginx
```

Get documentation about resources in Kubernetes

`kubectl explain RESOURCE`

- In Kubernetes we manage objects, or *resource types*
eg.
 - nodes
 - pods
 - namespaces
- The `explain` command displays documentation about specific kubernetes resources

Exercise: Ask kubectl about resources

- Use `kubectl explain` to find out about
 - nodes
 - namespaces
 - pods
 - services
- These are typical *resource types* in Kubernetes

```
$ kubectl explain node
KIND:      Node
VERSION:   v1

DESCRIPTION:
  Node is a worker node in Kubernetes. Each node will have a unique
  identifier in the cache (i.e. in etcd).

FIELDS:
  apiVersion    <string>
  APIVersion defines the versioned schema of this representation of an
  object. Servers should convert recognized schemas to the latest internal
  value, and may reject unrecognized values. More info:
  https://git.k8s.io/community/contributors/devel/api-conventions.md#resources
```

Configuring kubectl

- Minikube set up a configuration file for us
 - `~/.kube/config`
- Make sure we are using minikube's config

```
kubectl config use-context minikube
```

- Configuration file for kubectl
 - pass a configuration file with `--kubeconfig`

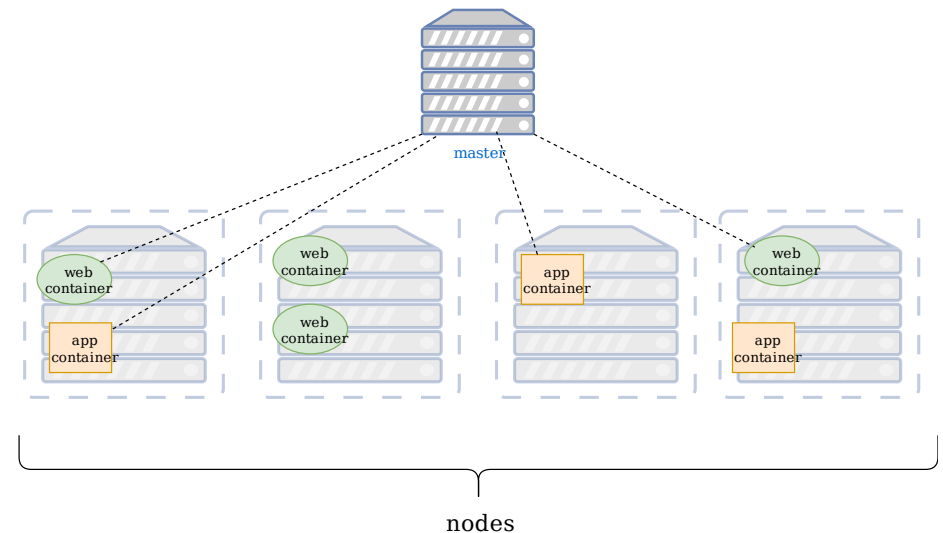
Kubernetes Resources

Kubernetes Resource Types

- Nodes
- Pod
- Deployment
- Namespaces
- Services
- Labels and Selectors

Nodes

- Nodes are where your containerised workloads will run
- No upper limit on number of nodes in a Kubernetes cluster
- Let's have a look at our *minikube* instance



Displaying Resources

`kubectl get RESOURCE`

- Retrieve information about kubernetes resources
 - eg. nodes

Exercise: Using `kubectl get`

- Use `kubectl get` to get info about current nodes

```
kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	master	6d	v1.10.0

- We currently only have one node named *minikube*
- The default stdout is minimal

Formatting output

- Many `kubectl` commands can output different data formats
 - `yaml`
 - `json`
- Pass `-o FORMAT` to command

Usage:

```
kubectl get [(-o|--output=)json|yaml|wide|custom-columns=...|  
custom-columns-file=...|go-template=...|  
go-template-file=...|jsonpath=...|jsonpath-file=...]
```

Exercise: Get formatted data about nodes

- Output node information in JSON

```
kubectl get nodes -o json
```

```
{  
  "apiVersion": "v1",  
  "items": [  
    {  
      "apiVersion": "v1",  
      "kind": "Node",  
      "metadata": {  
        "annotations": {  
          "node.alpha.  
          volumes.kub
```

- Quite a bit more information to process

Exercise: Process kubectl output

- It can be useful to pipe formatted output through other tools
 - For example **jq**
- Get a JSON list of node names with corresponding IP

```
kubectl get nodes -o json | jq '.items[] | {name: .metadata.name, ip: (.status.addresses[] | select(.type == "InternalIP")) | .ip}'
```

```
{  
  "name": "minikube",  
  "ip": "10.0.2.15"  
}
```


Running Workloads in Kubernetes

Running Containerised Workloads

```
kubectl create deployment name --image=IMAGE:TAG OPTIONS
```

- Create and run jobs in Kubernetes
- Example:

```
kubectl create deployment nginx --image=nginx
```

- Use **get** command to find out about container we just started

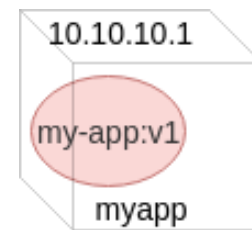
```
kubectl get containers
```

- Container isn't actually a resource type in Kubernetes

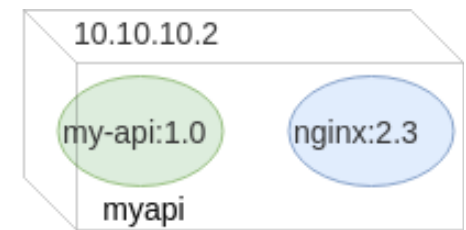
```
error: the server doesn't have a resource type "container"
```

Pods

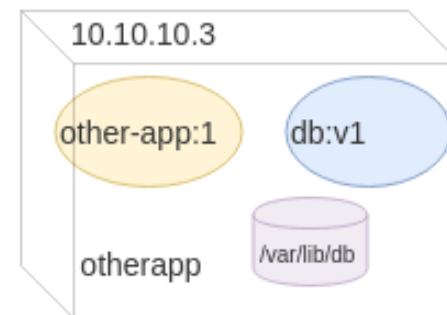
- Technically you do not run *containers* in Kubernetes
- The atomic *run unit* of Kubernetes is called a *Pod*
- A Pod is an abstraction representing group of ≥ 1 containers
 - images
 - network ports
 - volumes
- In this lesson we'll be using single container pods



Pod 1



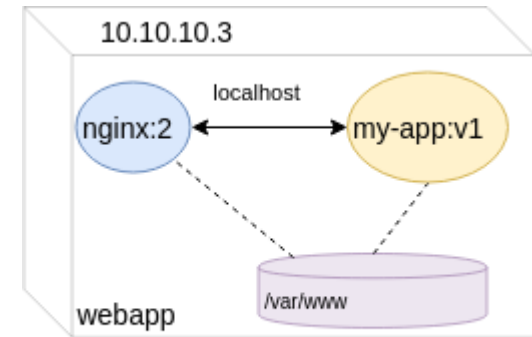
Pod 2



Pod 3

Pods

- Containers in a Pod share common resources
 - Network IP address
 - Mounted volumes
 - Always co-located and co-scheduled
- Containers within a Pod communicate via *localhost*



Exercise: Gather info about pods

- Use `kubectl get` to find info about running pods

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-5899c769f-ttt2x	1/1	Running	0	h

Running a Pod

- Let's run a *ping* command against Cloudflare's public DNS resolver

```
kubectl run pingpong --generator=run-pod/v1 --image alpine ping
```

```
deployment.apps "pingpong" created
```

- So, what is happening?

View logs for a pod

- The **logs** command behaves the same as with `docker logs`
- Query logs for pingpong

```
kubectl logs pingpong
```

Option	Description
-f, --follow	stream logs similar to <code>tail -f</code>
--tail	Specify how many lines from end to start with
--since	Get logs after a timestamp

Watching pods

- The **-w** option to kubectl is like the Linux **watch** command

```
kubectl get pods -w
```

- In another window run the following:

```
kubectl delete pod/pingpong
```

Scheduling Pods

```
kubectl run --schedule="*/5 * * * * " ...
```

- The `--schedule=` option takes a cron-like time pattern
- Creates a type of pod that runs periodically at assigned times
- In other words, a cronjob

```
kubectl run pi --generator=run-pod/v1 --schedule="0/5 * * * ?" --image=perl --restart=OnFailure -- perl -M
```

This pod calculates the value of PI to 2000 places every 5 minutes

Services

Routing traffic to pods

- Earlier we started an nginx pod
- At the moment there is no way to reach it

```
kubectl get pods -o json | jq '.items[] | {name: .metadata.name, podIP: .status.podIP }'
```

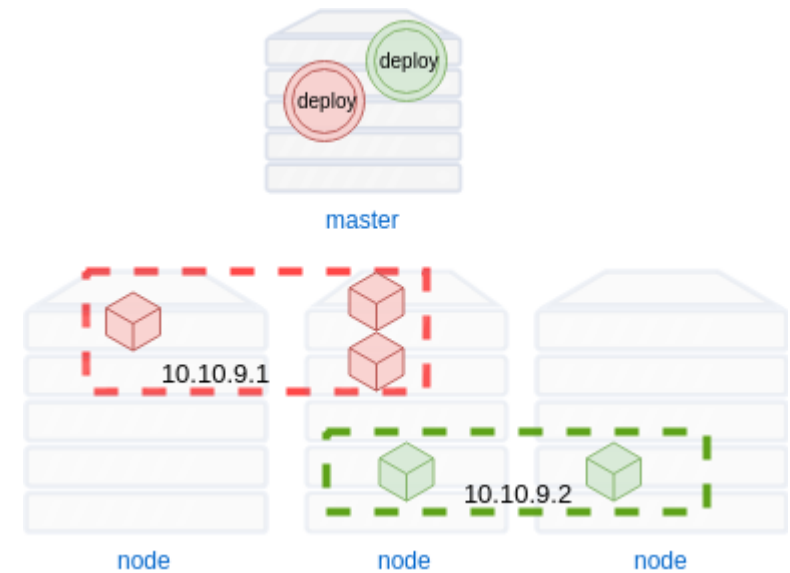
- Response

```
{  
  "name": "nginx-b848f798f-clrvd",  
  "podIP": "172.17.0.5"  
}
```

- The *podIP* is not reachable

Services

- A **service** defines logical set of pods and policy to access them
- ensure pods for a specific deployment receive network traffic



The expose command

```
kubectl expose -h
```

- Creates a service
- Takes the name (label) of a resource to use as a selector
- Can assign a port on the host to route traffic to our pod

Exercise: Expose nginx workload

- Need to create a service which
 - point to the *nginx* deployment
 - maps requests to port on nginx pod (port 80)
 - opens a port that is visible on our machine

```
kubectl expose deployment nginx --type=NodePort --port=80
```

```
service/nginx exposed
```

Get list of available services

- Get list of services

```
kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	106m
nginx	NodePort	10.110.58.243	<none>	80:#####/TCP	65m

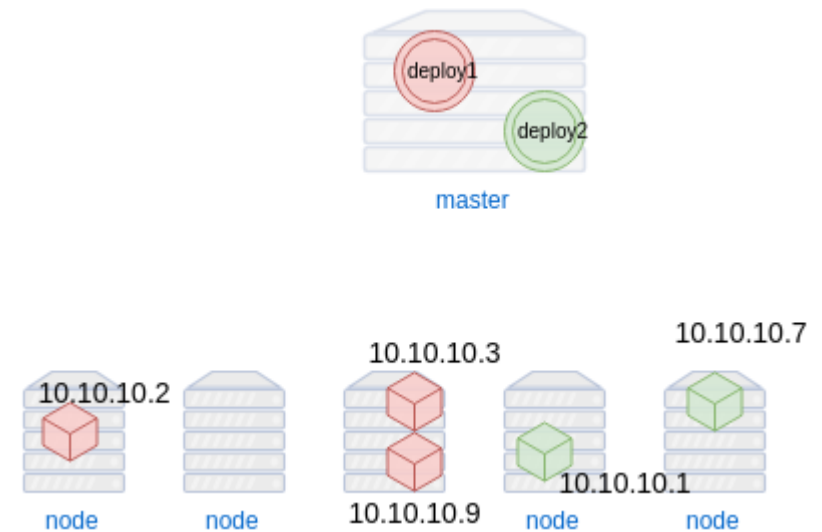
- Note the port that appears where the highlighted area is
- Open in your browser

```
firefox 192.168.99.100:#####
```

- Application should be exposed on the minikube IP at highlighted port

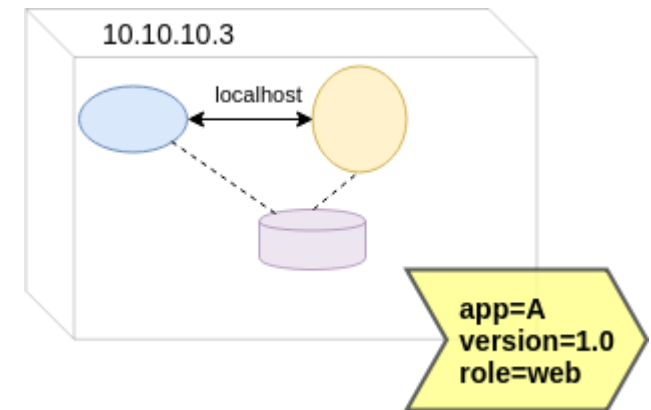
Networking Pods

- Each Pod in k8s has its own IP
 - even on same node
- Pod IPs never exposed outside cluster
- Pod IPs change often
 - updates/rollbacks
 - routine health maintenance
- Need a way to reliably map traffic to Pods



labels & selectors

- labels are key/values assigned to objects
 - pods
- labels can be used in a variety of ways:
 - classify object
 - versioning
 - designate as production, staging, etc.



Labels & Selectors

- Earlier we created a pod with label: **name=nginx**

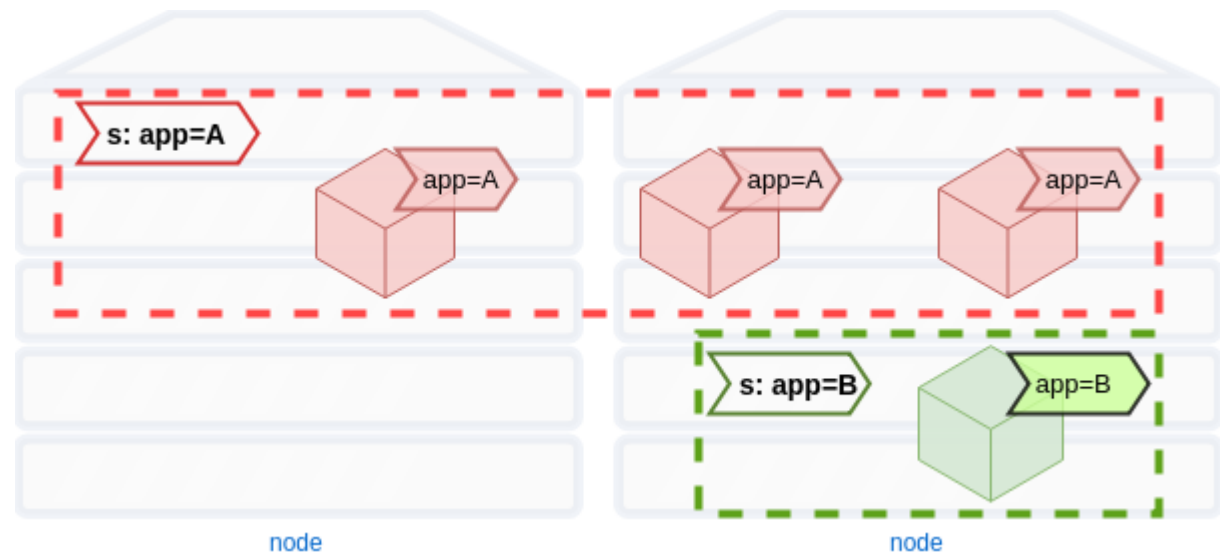
```
kubectl create deployment nginx --image=nginx
```

- Then we created a service that maps requests to the selector **name=nginx**

```
kubectl expose deployment nginx --type=NodePort
```

Matching Services and Pods

- Labels provide means for *services* to route traffic to groups of pods
- Services route traffic to Pods with certain label using *Selectors*

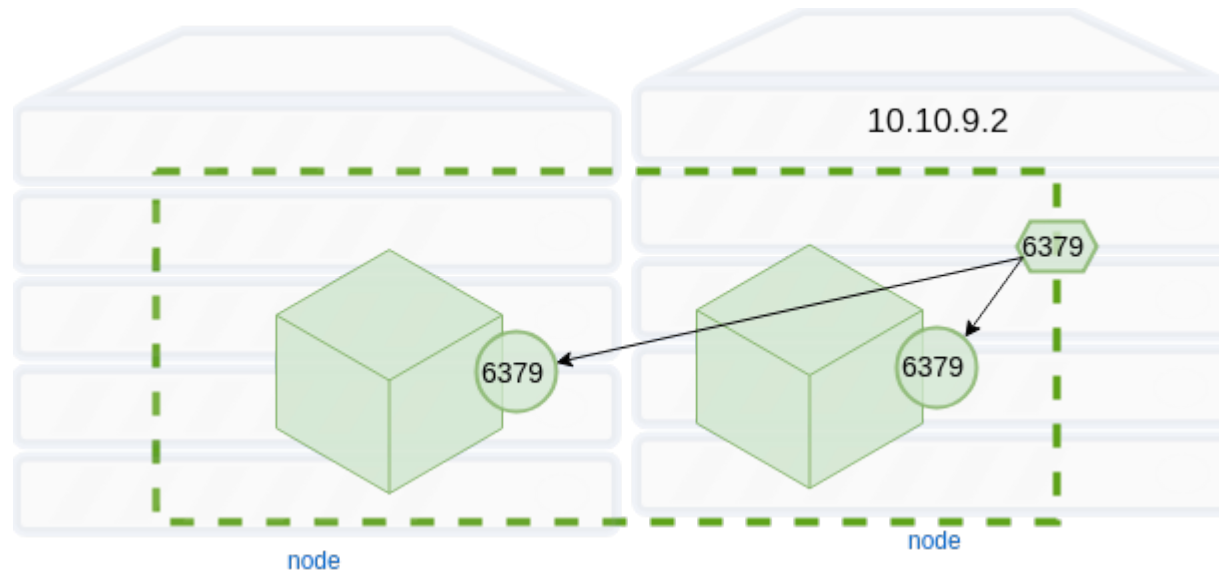


Service types

- *ClusterIP*
 - Exposes workload on an internal IP in the cluster
- *NodePort*
 - Expose workload on each node (assumes each node has public IP)
- *LoadBalancer*
 - Expose workload through single public IP

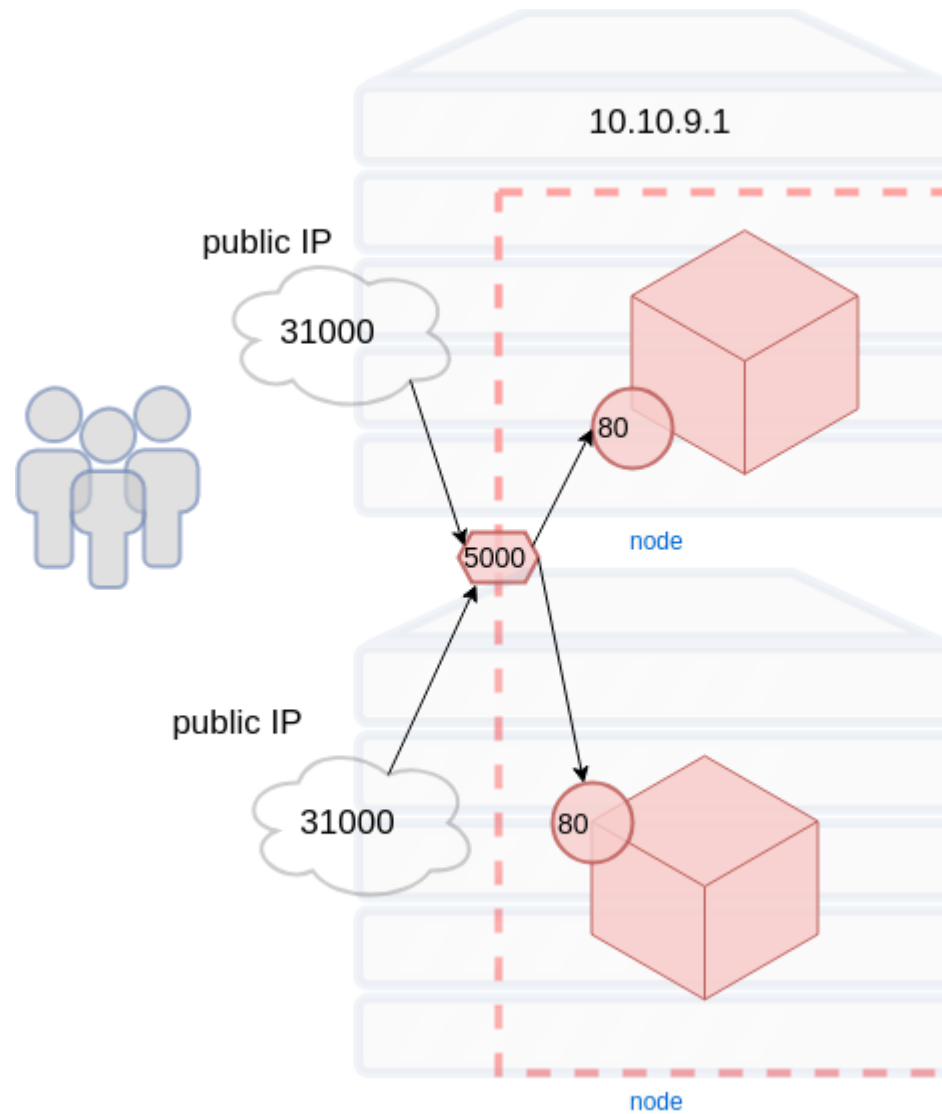
ClusterIP Service

Exposes the Service on an internal IP in the cluster



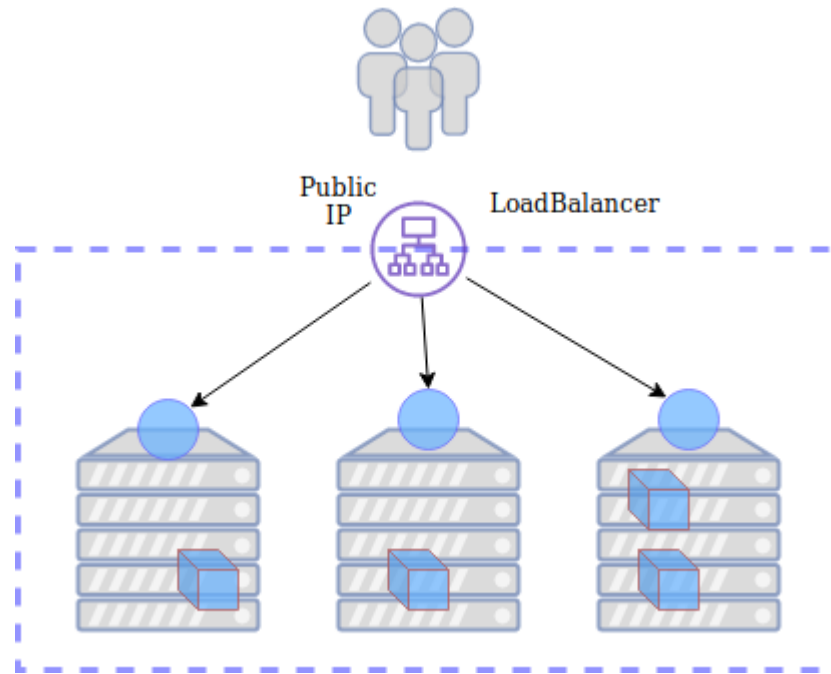
NodePort Service

Expose port on each node in cluster



LoadBalancer Service

Use load balancer to route traffic to service

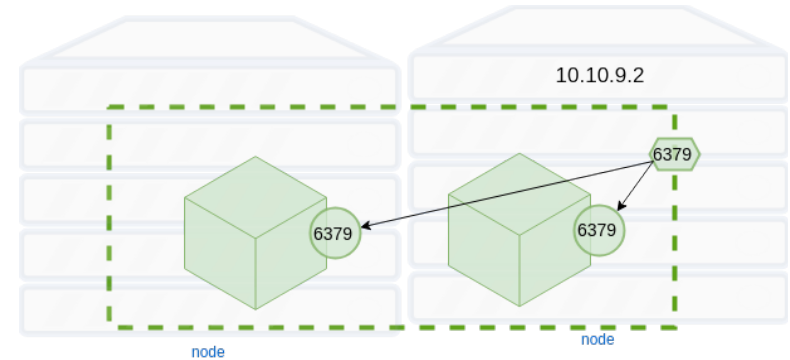


Service Specification Files

- It is possible to expose a Pod with a Service using `kubectl expose` command line
- For most applications, common to define *service specification* files
- A YAML or JSON file that defines a service

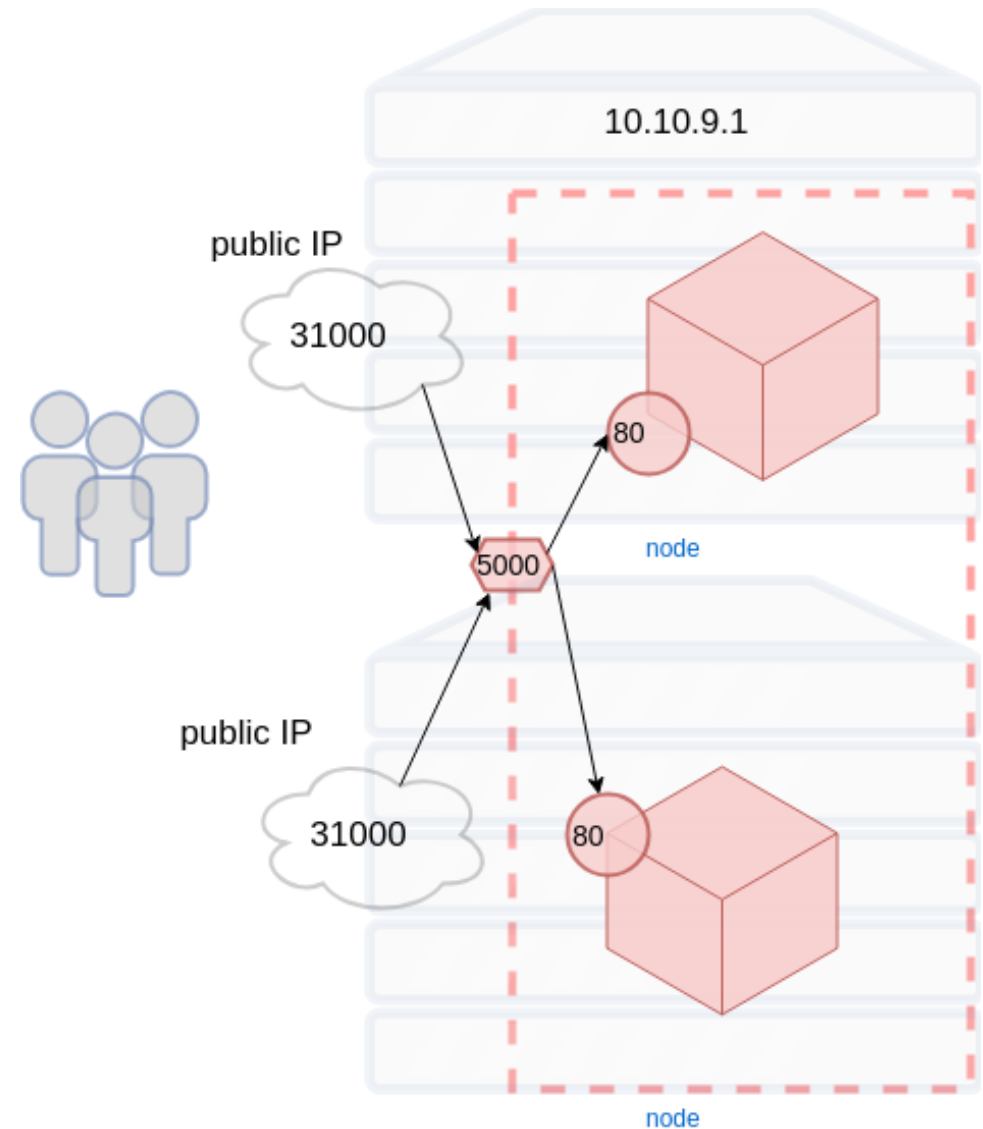
ClusterIP Spec File

```
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  type: ClusterIP
  ports:
    - port: 6379
      targetPort: 6379
  selector:
    app: redis
```



NodePort Spec File

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  name: "web-service"
  type: NodePort
  ports:
    - port: 80
      targetPort: 5000
      nodePort: 31000
  selector:
    app: app
```



Exercise: View service spec

- The `kubectl get` command can be used to get spec for a service or other objects
- Use it to view specification for nginx service

```
kubectl get svc nginx -o yaml
```

- Output spec to a file

```
kubectl get svc nginx -o yaml > nginx-service.yml
```

Exercise: Create service using spec file

- The spec file generated earlier can be used to create a service
- Delete the *nginx* service

```
kubectl delete svc nginx
```

- Create service using file

```
kubectl create -f nginx-service.yml
```

- Check that the service is up and the site is functional

```
kubectl get svc
```

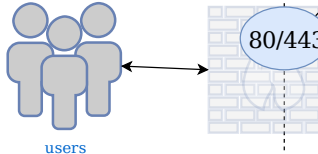
Exercise: Edit a spec inline

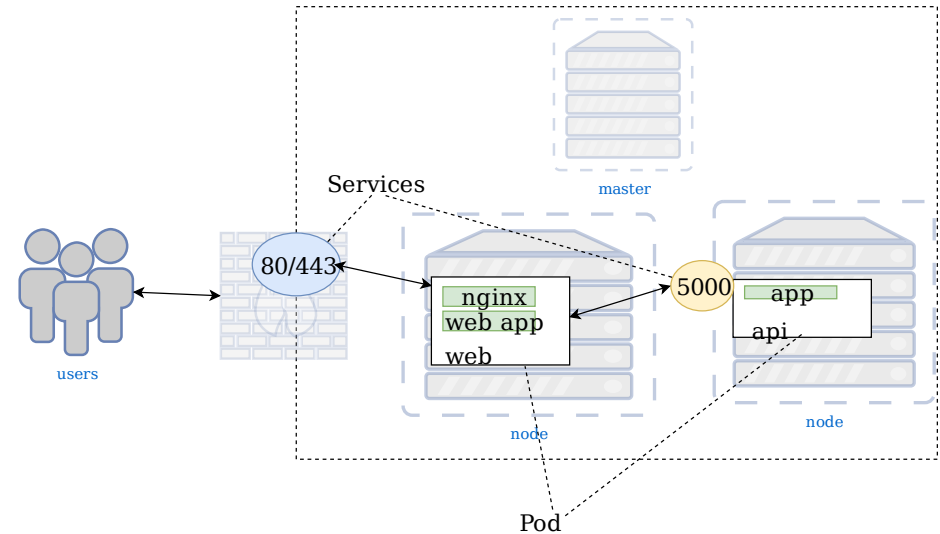
- It's also possible to edit a spec inline

```
kubectl edit service nginx
```

- This will open up an editor (eg. vim) so you can edit the service
- Try raising the value for the NodePort
 - Port must be between 30000 and 32767

Connecting an Application

- Functioning application depends on
 - Deployment
 - Pods
 - Services
 - Labels, selectors
- 
- The diagram illustrates a network interaction. On the left, three stylized human figures are grouped together, with the word 'users' written in blue below them. A double-headed arrow connects this group to a square area on the right that has a light blue brick pattern. Inside this square is a light blue circle containing the text '80/443'. A dashed line runs vertically along the right edge of the diagram.

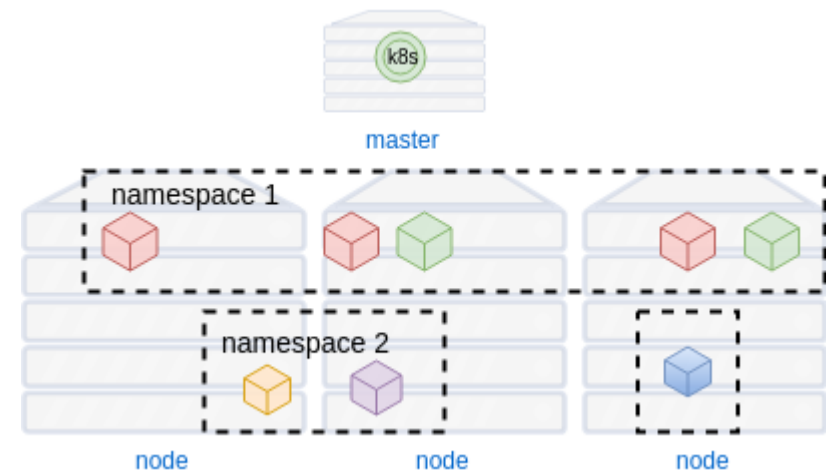


Pod

Namespaces

Namespaces

- A way to partition a Kubernetes cluster for different applications
- Partition physical cluster into virtual clusters



Exercise: Get list of namespaces

- Use `kubectl` to get list of namespaces

```
kubectl get namespace
```

NAME	STATUS	AGE
default	Active	8m44s
kube-node-lease	Active	8m47s
kube-public	Active	8m47s
kube-system	Active	8m47s

Namespaces in Kubernetes

- Kubernetes has several namespaces out of the box
 - default
 - Unless otherwise specified, objects will be created or queried here
 - kube-public
 - Reserved for cluster usage for resources that should be visible throughout cluster
 - kube-node-lease
 - Added in 1.14; a new way to implement node heartbeats
 - kube-system
 - Reserved for Kubernetes control

Operating in specific namespace

kubectl -n namespace COMMAND RESOURCE

- Specify a namespace with -n <namespace> flag

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

NAME	READY	STATUS	RESTART
etcd-minikube	1/1	Running	0
kube-addon-manager-minikube	1/1	Running	3
kube-apiserver-minikube	1/1	Running	0
kube-controller-manager-minikube	1/1	Running	0
kube-dns-86f4d74b45-nqx7c	3/3	Running	12
kube-proxy-z4qq5	1/1	Running	0
kube-scheduler-minikube	1/1	Running	3
kubernetes-dashboard-5498ccf677-ccvnx	1/1	Running	7
storage-provisioner	1/1	Running	8

kube-system namespace

- kube-system is used for the control plane
- Pods in this namespace *are* Kubernetes
 - etcd (eht-see-dee) simple storage for k8s
 - kube-apiserver The API server
 - kube-controller-manager and kube-scheduler
 - kube-dns Manages internal DNS
 - kube-proxy (on each machines) manages port mappings

Creating namespaces

`kubectl create namespace NAME`

- The create command can be used to create a new namespace
- Let's create a namespace

```
kubectl create namespace cats
```

- Verify that the new namespace exists

```
kubectl get ns
```

NAME	STATUS	AGE
cats	Active	2h
default	Active	2h
kube-public	Active	2h
kube-system	Active	2h

Create application in namespace

- Let's spin up an application in our new namespace

```
kubectl -n cats create deployment cat-app --image=heytrav/cat-of-the-day:v1
```

- Query state of pod/deployments in *cats* namespace

```
kubectl -n cats get all
```

NAME	READY	STATUS	RESTARTS	AGE
cat-app-b848f798f-clrzd	1/1	Running	0	2h

Expose our new application

- As with nginx example, we need a service to route requests to pods
- Let's create a **LoadBalancer** service to serve on port 80
- First we need minikube to open a tunnel

```
minikube tunnel
```

- Next we create a LoadBalancer service

```
kubectl -n cats expose deployment cat-app --type=LoadBalancer --port=80
```


Exercise: LoadBalanced application

- Watch available services to get IP

```
kubectl -n cats get svc -w
```

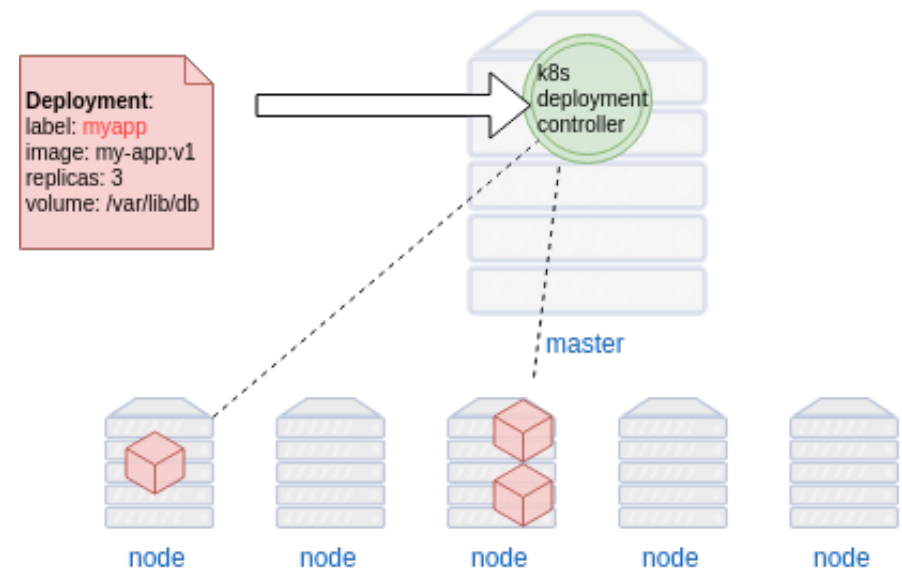
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
cat-app	LoadBalancer	10.100.25.215	10.100.25.215	80:30621/T

- Go to web application by putting the IP under EXTERN
IP in your browser

Deployments

Deployments

- A declarative configuration
- Tell Kubernetes the *desired state* of an application
 - which image(s) to use for an application
 - number of *replicas* to run
 - network ports
 - volume mounts
- The *deployment controller* changes cluster from actual to desired state



Maintaining Deployment State

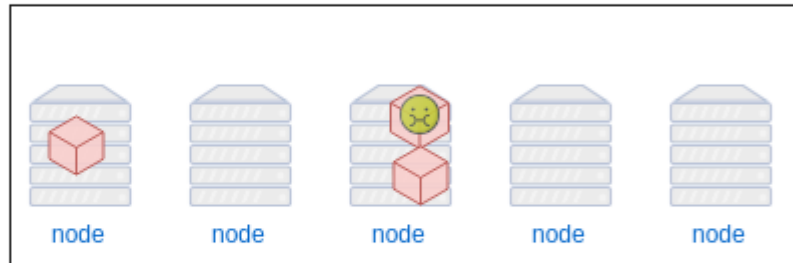
- Kubernetes continuously monitors state of application
- Self-healing
 - Replace unhealthy instances
 - Redistribute instances if node goes down
- Periodically culls and respawns instances

Maintaining Instance Health

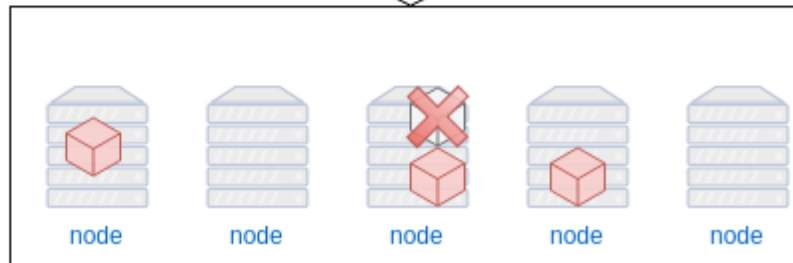
Deployment:
label: **myapp**
image: my-app:v1
replicas: 3
volume: /var/lib/db



Unhealthy Instance



End State



Recovery

Deployment:
label: myapp
image: my-app:v1
replicas: 3
volume: /var/lib/db



master

Node Down



End State

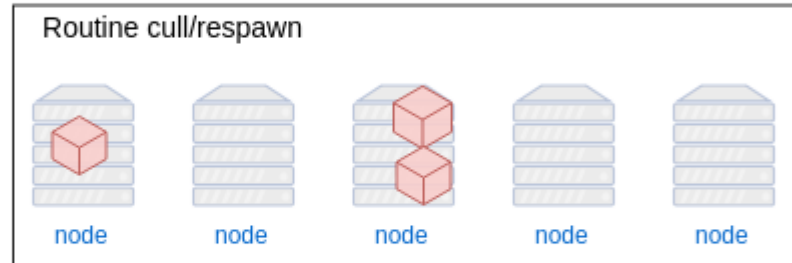


Routine Cull and Respawn Instances

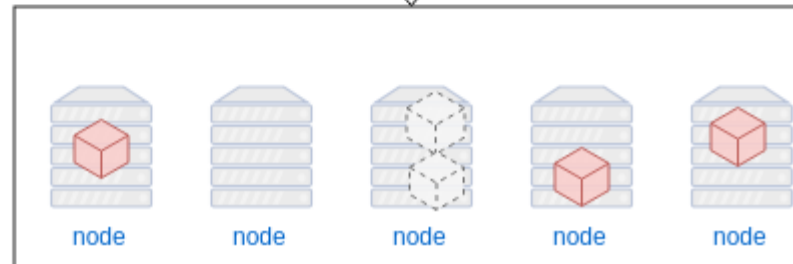
Deployment:
label: **myapp**
image: my-app:v1
replicas: 3
volume: /var/lib/db



Initial State



End State



Managing Deployments

- A *Deployment* can be modified at any time
 - *scaling*
 - changing number of replicas
 - *update*
 - change image for all instances
 - *rollback*
 - roll back to a previous version of application
- Kubernetes replication controller adapts to new desired state

Resizing resources

kubectl **scale** **RESOURCE** **OPTIONS**

- Set a new size for a resource
 - Deployment
 - ReplicaSet
 - Replication Controller
 - StatefulSet
- Specify preconditions
 - --current-replicas
 - --resource-version

Exercise: Scale number of replicas

- Increase the number of replicas (pods) for the *cat-of-the-day* application

```
kubectl -n cats scale deployment cat-app --replicas=4
```

- Query pods

```
kubectl -n cats get pods
```

- Try varying number of replicas up and down

Configuring existing resources

kubectl **set** **SUBCOMMAND** **OPTIONS**

- Make changes to existing application resources
- subcommands:
 - **env**: Update environment variables
 - **image**: Change image in a particular Pod
 - **resources**: Update resource limits on objects with Pod templates
 - **selector**: Set selector on a resource

Exercise: Update *Cat of the Day*

- Update the *cat-app* application

```
kubectl -n cats set image deployment cat-app cat-of-the-day=heytrav/cat-of-the-day:v2
```

- Refresh the website periodically

Rollout History

- It is easy to look over the revision history of your rollouts

```
kubectl rollout history deployment/cat-app -n cats
```

```
deployments "cat-app"  
REVISION    CHANGE-CAUSE  
1           <none>  
2           <none>
```

Rolling back

- It may be necessary to roll an application back to a previous version
 - update or new rollout is stuck
 - other bugs with application
- Use `rollout undo` to roll back to previous version
 - optionally specify revision with `--to-revision=`

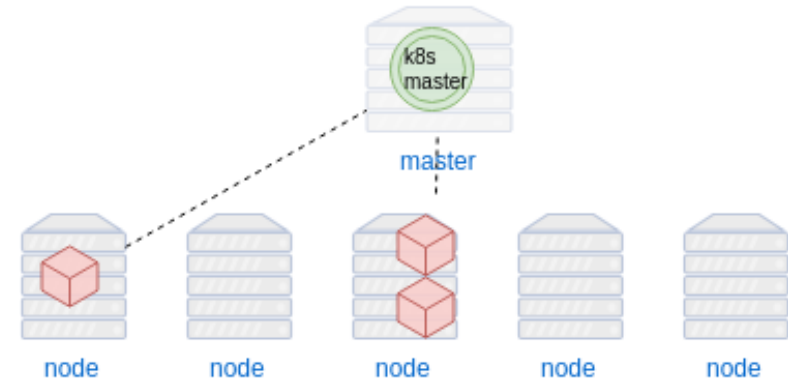
```
kubectl -n cats rollout undo deployment/cat-app \  
--to-revision=1
```

- Refresh the cat app site

Deployment Spec

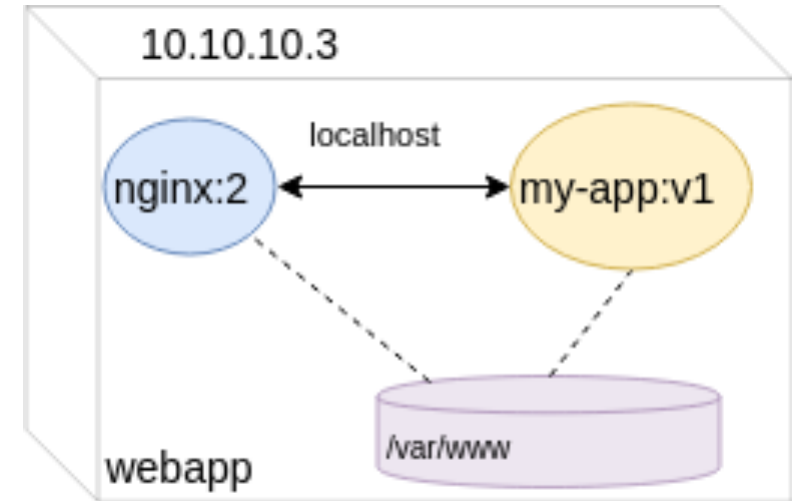
For more complicated deployments it is usually preferable to use a *Deployment Spec* file.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: webapp
    spec:
      .
      .
```



Deployment Spec

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - image: my-app:v1
          name: myapp
          volumeMounts:
            - mountPath: /var/www
              name: static-assets
        - image: nginx:2
          name: nginx
          ports:
            - containerPort: 80
          volumeMounts:
            - mountPath: /var/www
              name: static-assets
      volumes:
```



Deploy a Kubernetes Cluster

Setup a Kubernetes cluster

- Cloud providers
 - GKE
 - AWS EKS
 - Catalyst Cloud

OpenStack Command Line

- OpenStack provides a command line interface
- `openstack` command line script written in Python
- Installed on your machines in a virtual environment
 - should see `venv` in your command prompt
- Everything should be setup

Interacting with Catalyst Cloud

- Try out the command line client
- Query existing templates

```
openstack coe cluster template list
```

uuid	name
9c6e9df7-955a-465e-8460-e84e386624a0	kubernetes-v1.11.6-prod-20190130
4fcb04bd-22ba-4e1c-ab21-ff0339051d15	kubernetes-v1.11.6-dev-20190130
b1d124db-b7cc-4085-8e56-859a0a7796e6	kubernetes-v1.11.9-dev-20190402
cf337c0a-86e6-45de-9985-17914e78f181	kubernetes-v1.11.9-prod-20190402
967a2b86-8709-4c07-ae89-c0fe6d69d62d	kubernetes-v1.12.7-dev-20190403
f8fc0c67-84af-4bb8-89fb-d29f4c926975	kubernetes-v1.12.7-prod-20190403

Accessing your cluster

- Creating a cluster is easy, but it takes a while (~10 to 15 min)
- One has been created for you already
- Query information about your cluster

```
openstack coe cluster show -f json $PREFIX-k8s | jq '{"name": .name, "status": .health_status}'
```

- Response

```
{  
  "name": "trainpc-01-k8s",  
  "status": "HEALTHY"  
}
```

Interacting with your cluster

- As with minikube, you'll use `kubectl`
- First we must configure `kubectl` to talk to our new cluster
- `kubectl` can easily switch between different clusters or *contexts*

```
kubectl config use-context minikube  
kubectl get nodes
```

- Point it at the new cluster

```
kubectl config use-context default
```

Exercise: Verify Kubernetes Cluster

- Verify nodes

```
kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
trainpc-01-ivqn7i4nh66e-master-0	Ready	master	3h41m	v1.12.7
trainpc-01-ivqn7i4nh66e-minion-0	Ready	<none>	3h41m	v1.12.7
trainpc-01-ivqn7i4nh66e-minion-1	Ready	<none>	3h41m	v1.12.7

Exercise: Get JSON list of nodes with IPs

```
kubectl get nodes -o json | jq '.items[] | {name: .metadata.name, ip: (.status.addresses[] | select(.type == "ExternalIP")) | .ip}'
```


Summary

- We each have a functioning Kubernetes cluster
- Can use `kubectl` to maintain multiple clusters
- Set up the control plane

Management Dashboard

Management Dashboards

- A kubernetes cluster has many moving parts to keep track of
 - Control Plane
 - Workloads
 - Configs, Secrets
- Lots of commands to remember for `kubectl`
- Dashboards useful for managing as well as monitoring your cluster

Dashboard options

- Standard Kubernetes dashboard
 - As seen with `minikube` dashboard
 - Can setup using **instructions** in Catalyst Cloud documentation
- Third party alternatives
 - RancherOS Kubernetes Dashboard

RancherOS dashboard

- Complete management interface
- Manage multiple clusters
- Lots of additional features
 - manage multiple authentication backends
 - launch applications
 - monitoring

Access the RancherOS dashboard

- Open the management interface

```
open - rancher
```

- Click through the security warning
- You'll need to set an admin password and confirm the page url
 - just use *admin* or something simple
- Confirm the IP address
 - just click *Save URL*

Add Kubernetes Cluster

- Click on *Clusters* and *Add* a cluster
- Click on **Import** button on the right under *Import existing cluster*
- Enter a name in the *Cluster Name* field
 - i.e. *sandbox*
- Click *create*

Add Kubernetes Cluster

- Copy the command in the box with:

```
curl --insecure -sfL ...
```

- Paste and execute into your terminal
- Click *Done* and select your new cluster from the list
- In the following screen select your cluster
- It will take a few seconds/minutes to register with the dashboard

The RancherOS dashboard

- Provides a main *Cluster* screen with some basic monitoring data (CPU, Memory, Pods)
- We can also manage our cluster here; for example
 - setup authentication (OAuth, LDAP, etc.)
 - create namespaces
 - scale deployments
 - drain nodes
- We will explore some of the features as we deploy our app in upcoming section

Add monitoring to our cluster

- In the top menu, select *Tools* and *Monitoring*
- Just scroll to bottom and click *Enable Monitoring*
- Navigate back to your main cluster dashboard
- Monitoring should be available in a couple minutes
- Initially there won't be much to see so we'll check back later

Explore dashboard

- Namespaces
- Nodes
- Tools

Launch an app

- On far left tab drop down select *default*
- In the next menu select *Apps*
- Catalogue displays apps available has Helm charts
- Launch a Wordpress application

Deploying to Kubernetes Cluster

Setting up the Voting Application

- Have a look at kubernetes specs for the vote app

```
cd ~/example-voting-app
```

```
ls k8s-specifications
```

- Folder contains specification files for
 - Deployments
 - Services

Creating a namespace

- In dashboard click *Projects/Namespaces*
- In the *Project Default* section, click *Add Namespace*
- On next page enter **vote** in *Name* field and click *Create*
- This creates a new namespace in your cluster
- Can confirm by running in the shell

```
kubectl get ns
```

Watch cluster

- A couple ways to watch what is happening
- In your terminal

```
watch kubectl -n vote get all
```

- In the dashboard
 - In *Project/Namespaces* menu
 - click on *Project: Default*
 - click on *Workloads* tab

Load Specification Files

```
kubectl apply -f specfile.yml
```

- The `apply` command *applies* a configuration to a specific resource
- The entire vote app is specified in yaml files

```
cd ~/example-voting-app  
kubectl apply -n vote -f k8s-specifications
```

- This tells Kubernetes to begin setting up containers
 - creates network endpoints
 - assigns Pods to replication controller
- When you run this, go back to the *watcher* terminal

Viewing Vote Website

- Once all containers are running you can visit the *vote* and *result* apps
- We need to know how to reach them
- Both deployments are exposed with *NodePort* services
- Can reach them if we know the IP of any node and the port the service is exposed on

Node IPs

- Use kubectl with -o wide to get node ips

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

NAME	EXTERNAL-IP
trainpc-01-*	202.49.243.126
trainpc-01-*	202.49.243.127
trainpc-01-*	202.49.243.128

- Get services

```
kubectl -n vote get svc  
vote Services
```

NAME	TYPE	CLUSTER-IP	PORT(S)	AGE
db	ClusterIP	10.108.228.228	5432/TCP	3h
redis	ClusterIP	10.107.101.100	6379/TCP	3h
result	NodePort	10.107.43.36	5001:31001/TCP	3h
vote	NodePort	10.104.244.69	5000:31000/TCP	3h

- Use any of the node ips with port to view website

Exercise: Scale number of replicas for vote

- Increase the number of replicas (pods) for the *vote* service
- In the terminal

```
kubectl -n vote scale deployment vote --replicas=9
```

- .. or try using dashboard UI
- Keep an eye on *watcher* terminal
- Try varying number of replicas up and down

Exercise: Update voting app

- Update the *vote* application with your image

```
kubectl -n vote set image deployment/vote \  
vote=YOURNAME/vote:v2
```

- Watch the *watcher* terminal
- Refresh the site several times while update is running

Summary

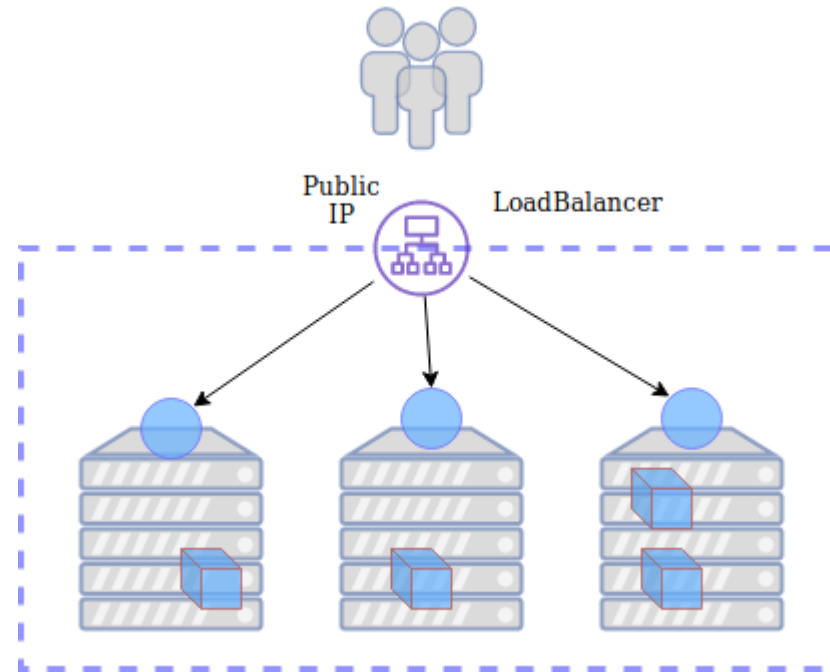
- In this section we deployed a microservice application
- kubectl can scale services up or down
- Deployment rolling updates ensure that the application updates seamlessly with zero downtime

LoadBalancer Service

Problems with NodePort

- Using a NodePort service is ok for test purposes
- Not practical for an actual website:
 - eg. visit my website at <http://202.49.243.126:31000>
- Prefer to use
 - fixed domain
 - standard ports (i.e. 80, 443)

LoadBalancer Service



- LoadBalancer Service is actually combination of
 - NodePort (>31000) on each node
 - Load balancer set up by your cloud provider

LoadBalancer Service

- Let's create a load balancer service for the *vote* app

```
kind: Service
apiVersion: v1
metadata:
  name: loadbalanced-service
spec:
  selector:
    app: vote
  type: LoadBalancer
  ports:
    - name: http
      port: 80
      protocol: TCP
```

- Save this as `loadbalancer.yml`
- Apply with `kubectl`

```
kubectl -n vote apply -f loadbalancer.yml
```

Using a LoadBalancer service

- Check available services to see when load balancer is ready

```
kubectl -n vote get svc -w
```

- It takes a few minutes while your provider provisions the load balancer
- Once finished a public IP will appear under **EXTERNAL-IP**

```
kubectl -n vote get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
db	ClusterIP	10.254.70.29	<none>	5432/TCP
loadbalanced-service	LoadBalancer	10.254.167.242	202.49.243.158	80:31909/TCP
redis	ClusterIP	10.254.125.187	<none>	6379/TCP
result	NodePort	10.254.236.164	<none>	5001:31001/TCP
vote	NodePort	10.254.234.53	<none>	5000:31000/TCP

Advantages of LoadBalancer

- With a **LoadBalancer** type service you can now access your website on the new IP on a standard port (eg. 80 or 443)
- Can setup DNS with your domain name of choice
 - eg. Visit my website <http://my-vote-app.nz>

Disadvantages of a LoadBalancer

- We can only expose one workload
 - Have a LoadBalancer for *vote* pods
 - Need a separate LoadBalancer for the *result* pods
- Load balancers are expensive
- Separate public IP so need a second domain

Remove load balancer

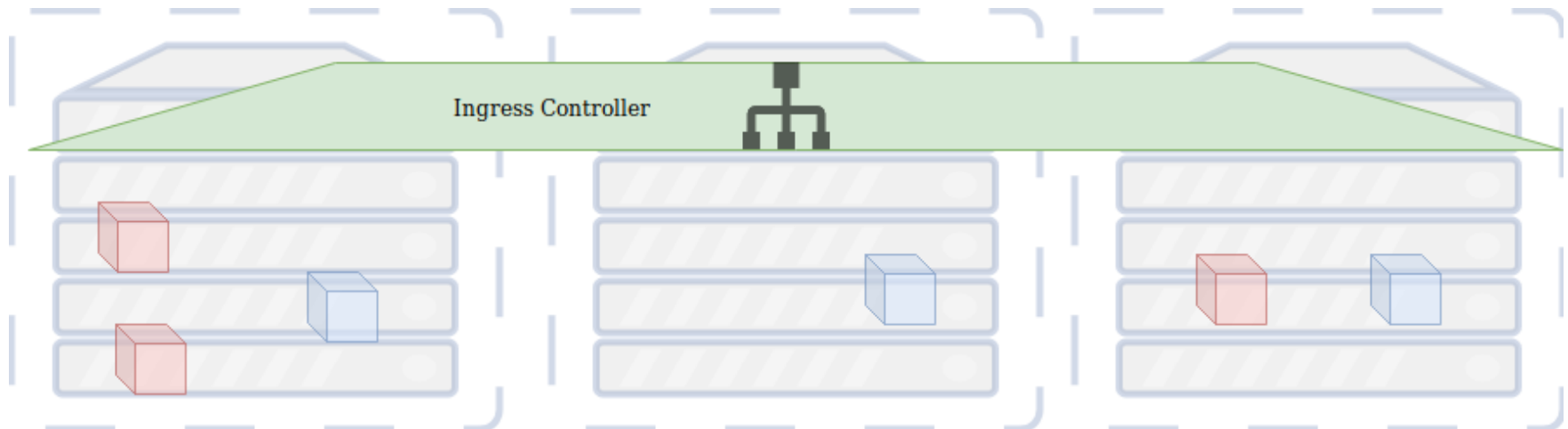
- In the next section we'll look at an alternative to using the LoadBalancer service
- For now let's delete the load balancer

```
kubectl -n vote delete -f loadbalancer.yml
```

Ingress Controllers

Ingress Controller

- A Kubernetes resource type
- Alternative way to expose workloads in a cluster
- Runs in your cluster and routes traffic to container workloads



Setting up Ingress

- First we need setup Helm and Tiller on your cluster

```
setup-tiller
```

- Use helm to install an ingress controller

```
helm install stable/nginx-ingress --name my-nginx
```

- should generate a bunch of output

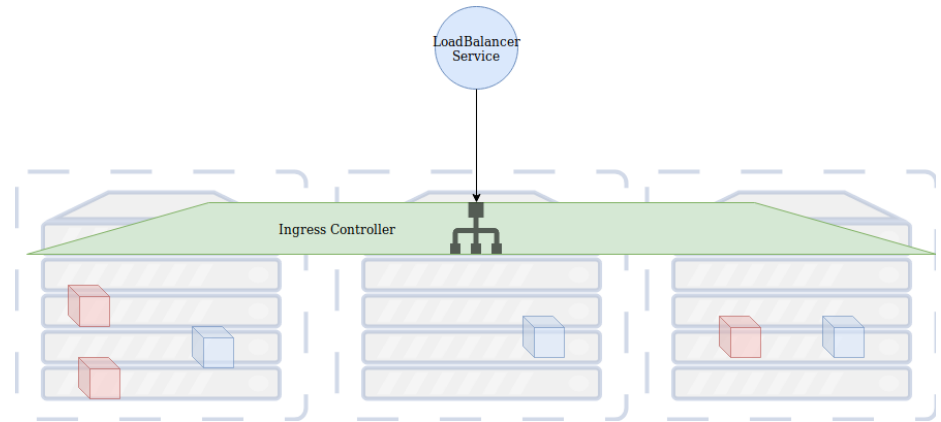
- Watch cluster to see when new LB is created

```
kubectl get services -o wide -w my-nginx-nginx-ingress-cont
```

- Note the **EXTERNAL-IP** that is created

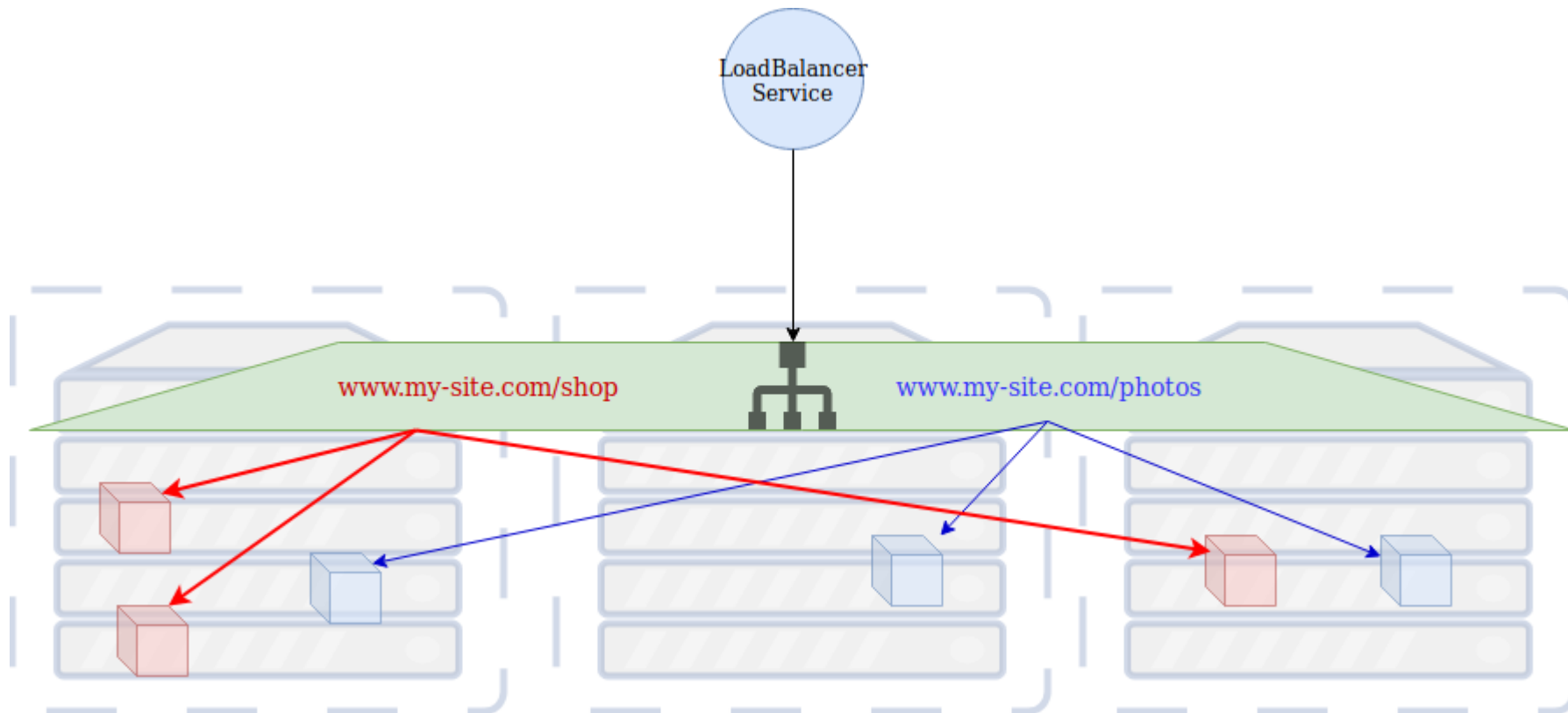
Ingress and LoadBalancer

- You still need a LoadBalancer to route traffic to nodes
- The advantage of Ingress is how it can route traffic to your applications



Ingress routing

Path-based routing



Exercise: Create our first ingress

- Let's run the cat app in our cluster

```
kubectl create ns cats  
kubectl -n cats create deployment cat-app --image=heytrav/cat-of-  
kubectl -n cats expose deployment cat-app --port=5000
```

- Let's also start an nginx pod and expose it on port 80

```
kubectl -n cats create deployment nginx --image=nginx  
kubectl -n cats expose deployment nginx --port=80
```

- Add **my-cats.com** to `/etc/hosts` with the external IP of your loadbalancer

```
xxx.xxx.xxx.xxx    my-cats.com
```

Expose our ingress using paths

- Put the following in a file called `cats-ingress.yml`

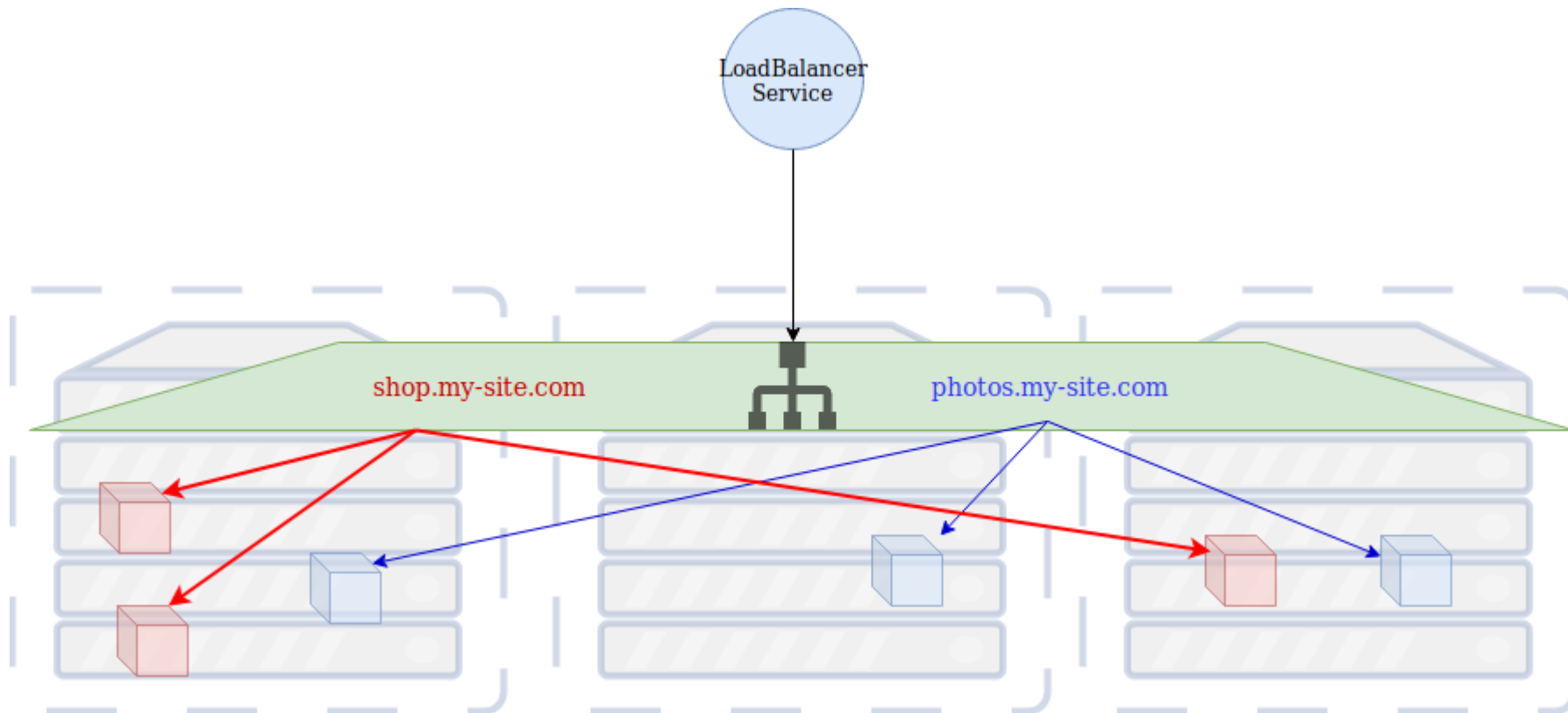
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: cat-ingress
  namespace: cats
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: my-cats.com
      http:
        paths:
          - backend:
              serviceName: cat-app
              servicePort: 5000
            path: /cats
          - backend:
              serviceName: nginx
              servicePort: 80
            path: /hello
```

- Create the ingress

```
kubectl create -f cats-ingress.yml
```

Ingress routing

domain-based routing



Define an ingress for vote app

- This can also be done with a spec file

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: vote-ingress
  namespace: vote
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
    - host: vote.my-app.com
      http:
        paths:
          - backend:
              serviceName: vote
              servicePort: 80
    - host: result.my-app.com
      http:
        paths:
          - backend:
              serviceName: result
              servicePort: 80
```

- Let's do it in the dashboard instead

Setup domain-based ingress

- First, add a couple more hosts to /etc/hosts

```
xxx.xxx.xxx.xxx    my-cats.com vote.my-app.com result.my-app.com
```

- In dashboard navigate to the *Loadbalancing* interface and click *Add Ingress*
- Enter a name in the *Name* field and select **vote** from the *Namespace* menu
- Select *Specify a hostname to use* and enter **vote.my-app.com**
- Leave *path* blank, select **vote** under *Workload* and enter port **80**

Setup domain-based ingress

- Click *Add Rule* and repeat similar steps as above, replacing *vote* with *result* where applicable
- When complete, click *Save*

Summary

- Ingress controllers are a versatile and cost-effective alternative to LoadBalancer service
- Can route traffic for multiple sites
 - path-based routing
 - subdomain-based routing

Maintaining clusters

Updating Nodes

- Important to maintain up-to-date stack
- Security updates and patches for underlying OS
- This section explores regular maintenance task on a k8s cluster
- It's good to regularly update machines
- Easiest way is to simply replace VMs on a regular basis

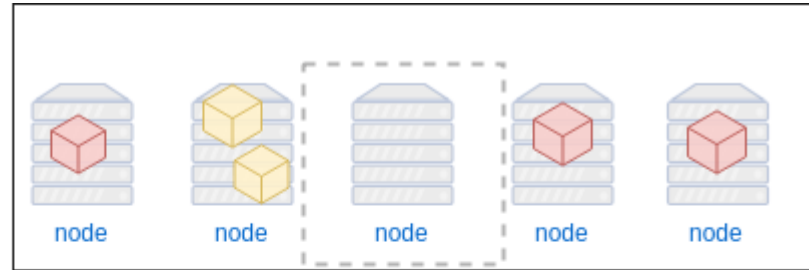
Draining nodes

kubectl **drain** **OPTIONS**

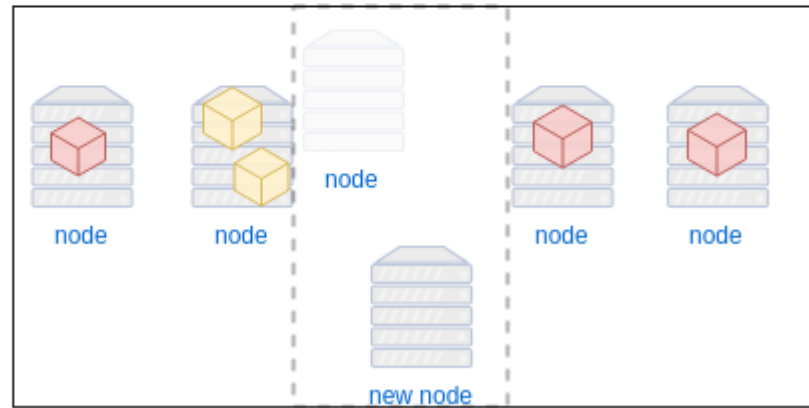
- Tell kubernetes to take a node out of service
- Node is *cordoned* off
- Pods will be respawned on other machines
- K8s will not restart any Pods until node is *uncordoned*

Drain Node

Drain/Cordon
off Node



Drop Old
Node/Build
replacement



Adding back a node

```
kubectl uncordon node
```

- Uncordon returns or adds a cordoned node back to service
- K8s will not immediately add pods to node
 - culling/respawning
 - scale operations

Uncordon Node

Uncordon
node



Cordon
and drop
reserve
node



Managing maintenance

- Dashboard provides some tooling for draining/uncordoning nodes
- Ideally your cloud provider will do this for you
 - Drain/Cordon
 - patching/replacing nodes
- cluster options
 - built in
 - trigger via API call

Summary

- Rebuilding nodes on a regular basis essential for security
- K8s provides means for coordinating stack maintenance
- Draining and rebuilding nodes easy to automate

The End