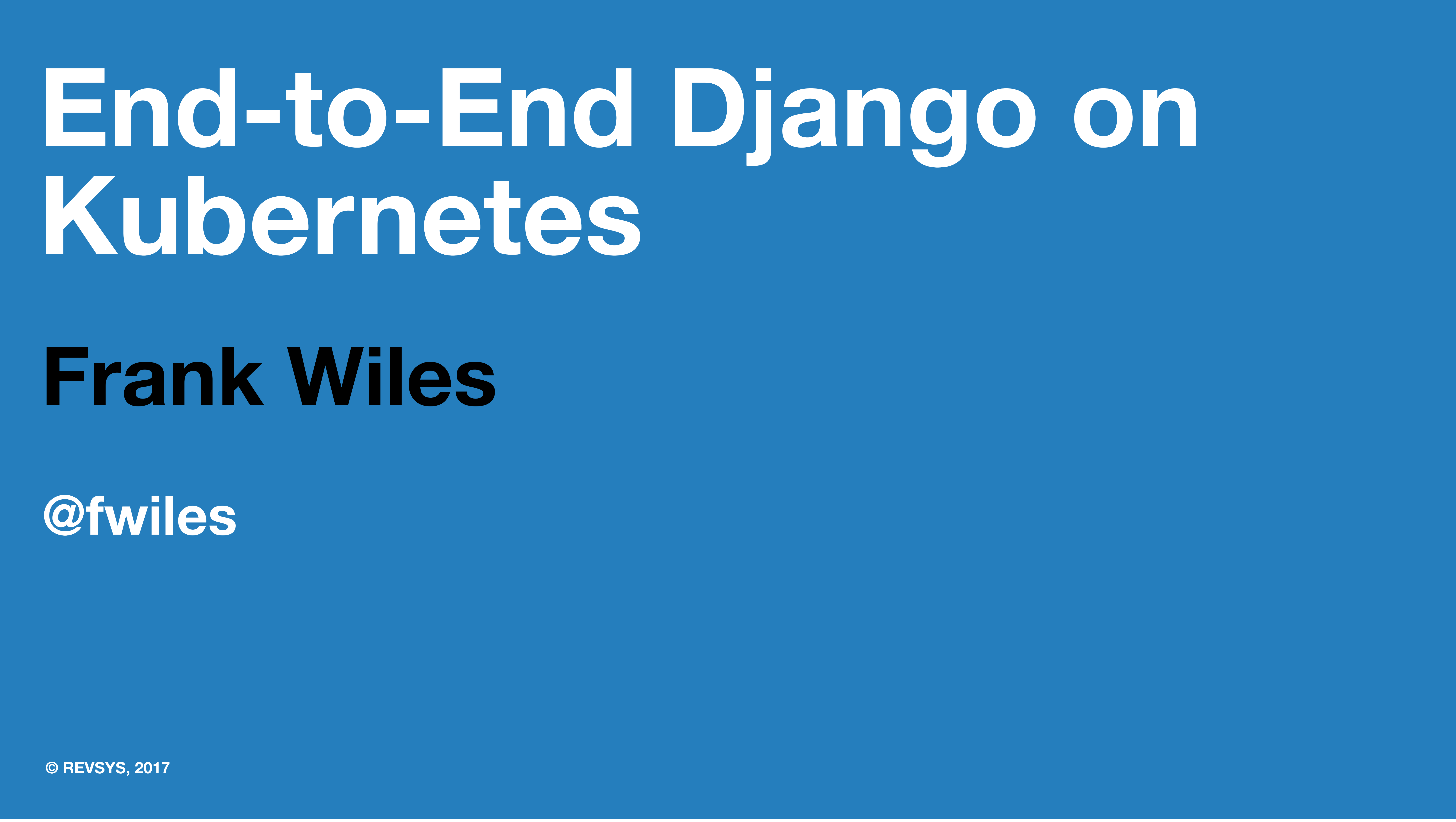


# End-to-End Django on Kubernetes

## Frank Wiles

@fwiles

Good afternoon everyone, thanks for having me.
This may come as a surprise to you, but...

# I am not Josh Berkus

I am not Josh Berkus
But the more I thought about it, the more I realized how you could be confused...

I mean Josh's name is on the program

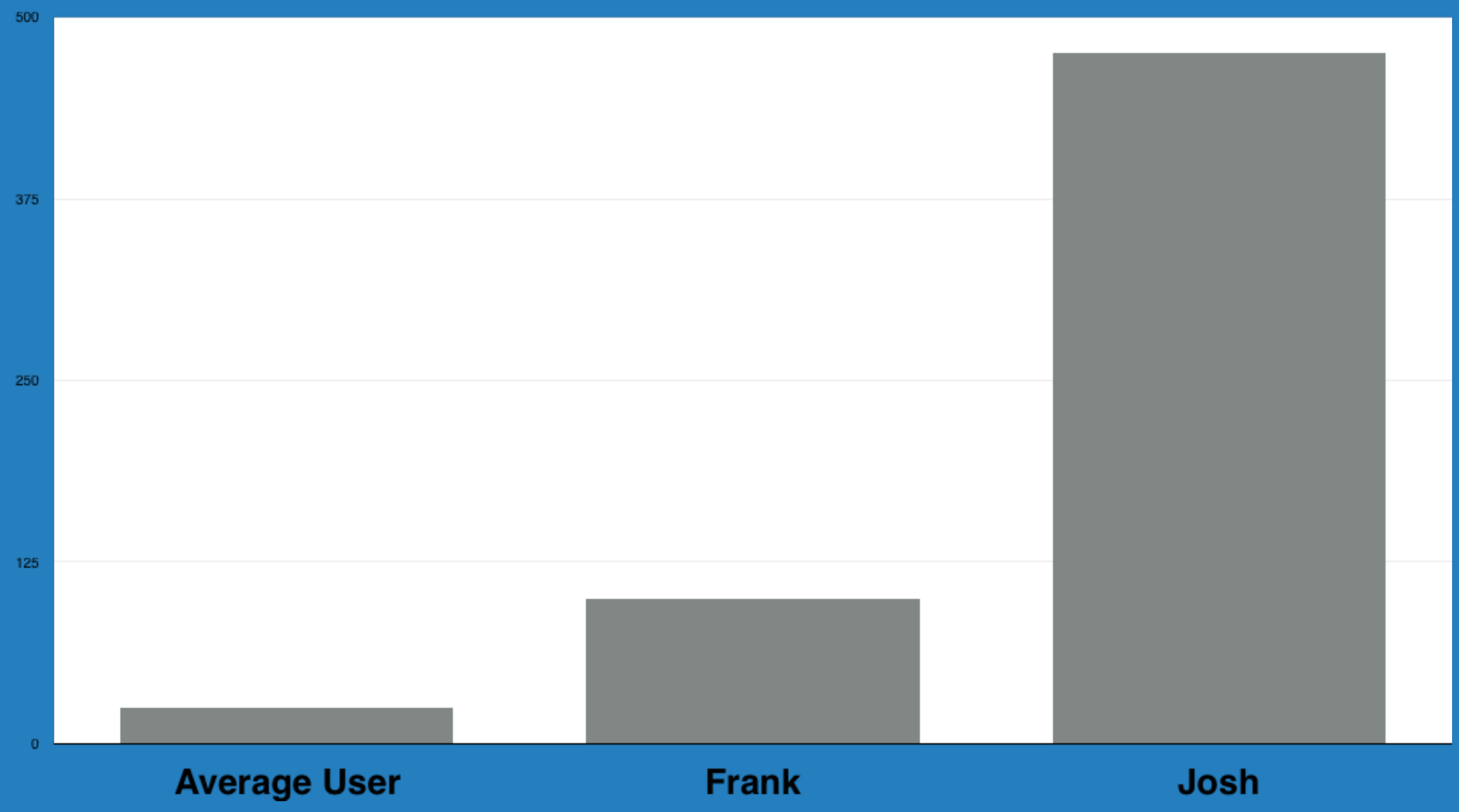
We both have beards

We both like Django, PostgreSQL, and Kubernetes

We both even have the **same damn glasses**

But we also have some differences

## PostgreSQL Knowledge



Over here on the left we have the average PostgreSQL user's knowledge.
I know a little bit more, but then over here on the right it's clear Josh knows a ton more than me.
You're probably asking yourself what the hell this has to do with anything, but
there is one final important difference that pertains to this talk.

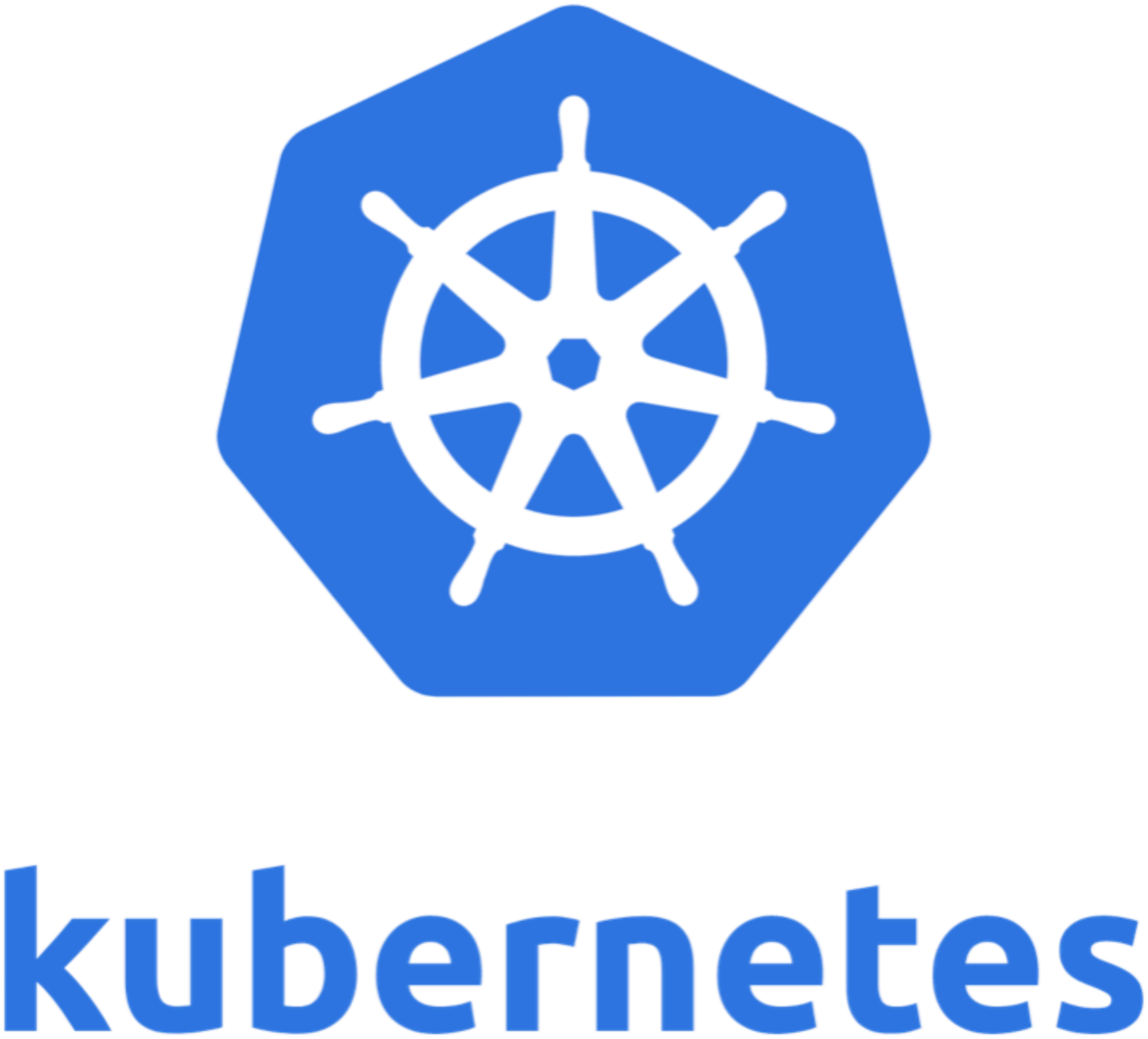
# Back works ok

My backing is working pretty good today (stretch)

# Not so much

Josh's, not so much.

Which is why I'm up here to talk to you about Django and Kubernetes.
 See, Josh managed to hurt his back last week while making the awesome speaker gifts for DjangoCon.
 And the DjangoCon team asked me to fill in for him.
Luckily, I've been using Django on Kubernetes with clients for awhile so I didn't even need to change the topic!



Speaking of the topic, we should probably get to that. Kubernetes is arguably the best and most popular container orchestration system in use today. Before we dive into things too deeply we need to get some terminology straight.

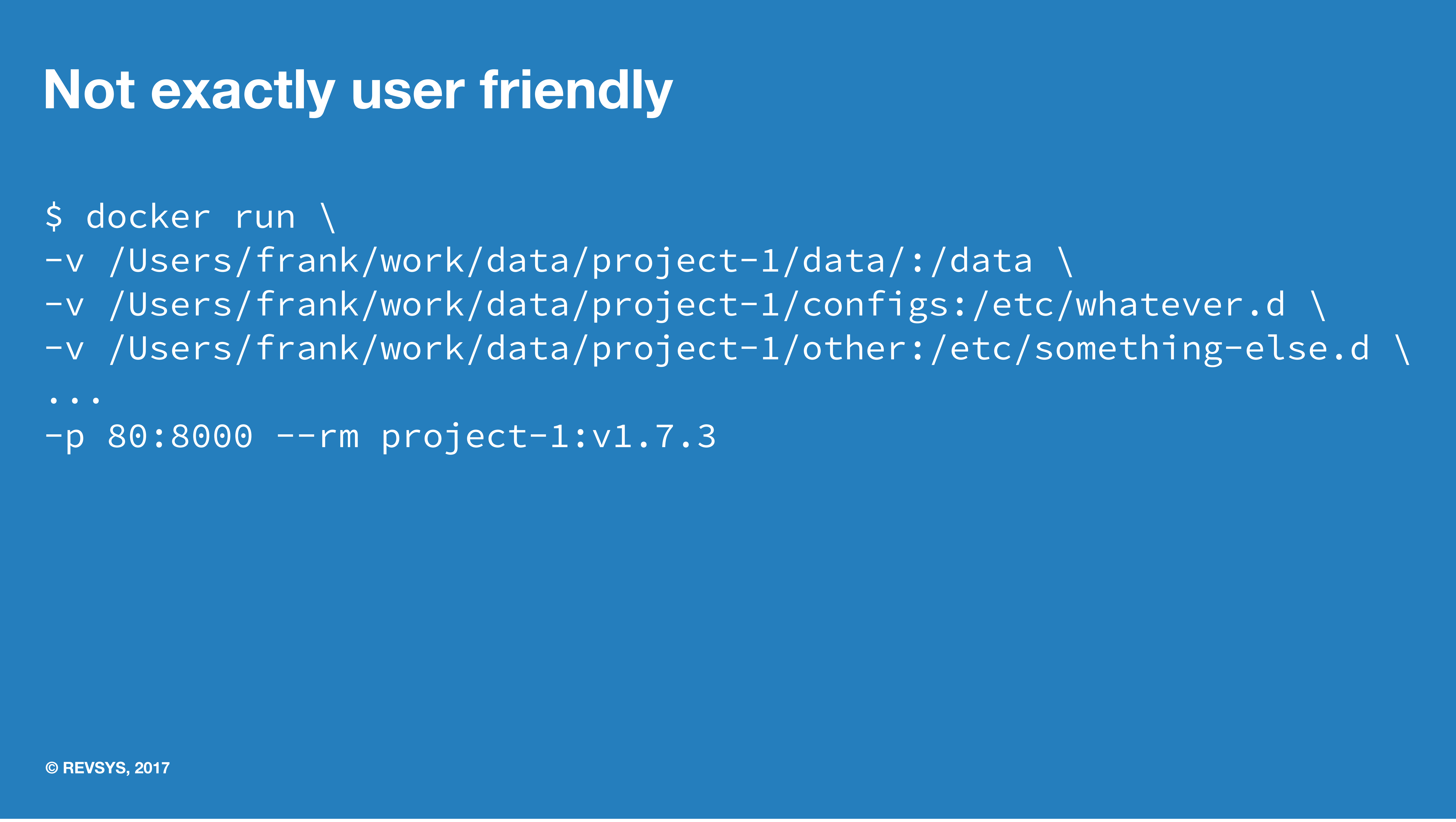
**First off the name, Kubernetes means...**

1. **Greek for ship captain**
2. **Google learned it's lesson naming things after Go**
3. **All of the above**

If you picked number 3 you're correct!

# Containers are great but...

The are great at running our application process
and packaging up all of it's dependencies
but on their own, they can be difficult to work with



# Not exactly user friendly

```
$ docker run \
-v /Users/frank/work/data/project-1/data/:/data \
-v /Users/frank/work/data/project-1/configs:/etc/whatever.d \
-v /Users/frank/work/data/project-1/other:/etc/something-else.d \
...
-p 80:8000 --rm project-1:v1.7.3
```

Outside of a single container, things get messy. So this is why container orchestration services like docker-compose, docker swarm, mesos, AWS Container Service, and kubernetes exist

# What is container orchestration anyway?

> Event loops are used by the Kubernetes components to reconcile things between the local machines and the desired cluster state.
> — Kelsey Hightower

# It's really a control loop

> **In applications of robotics and automation, a control loop is a non-terminating loop that regulates the state of the system. In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state.**
> **— Kubernetes Documentation**

We simply define how we want things to look. Which apps are deployed and who they can communicate with and Kubernetes works to make that vision a reality for us. I'm not going to lie to you and say Kubernetes is super easy to learn. It's a big complicated system.

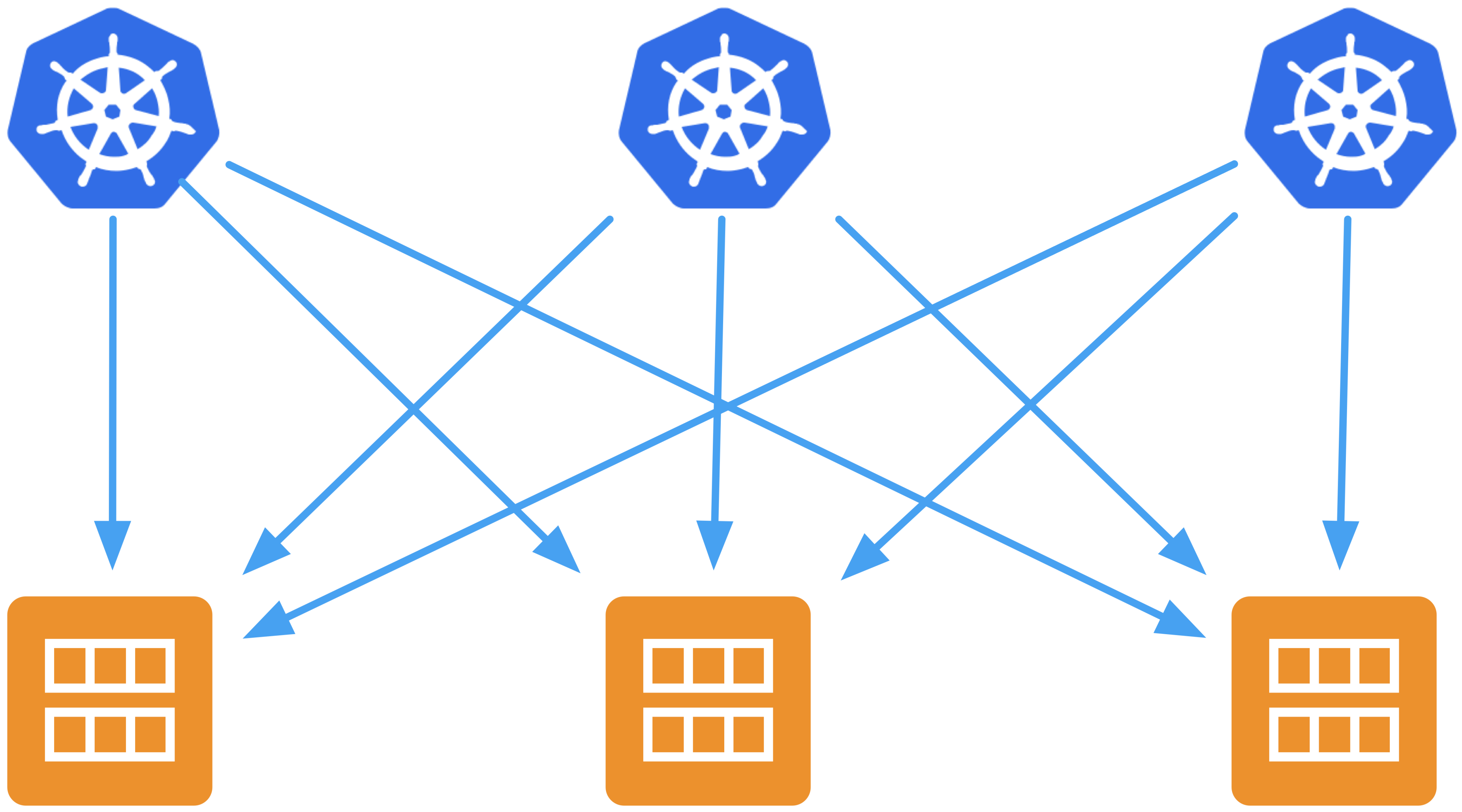
It's a bear. A big scary bear. We're not going to learn all about it in 40 minutes, but my goal is to change your impression of it from this to...

..this

# Terminology

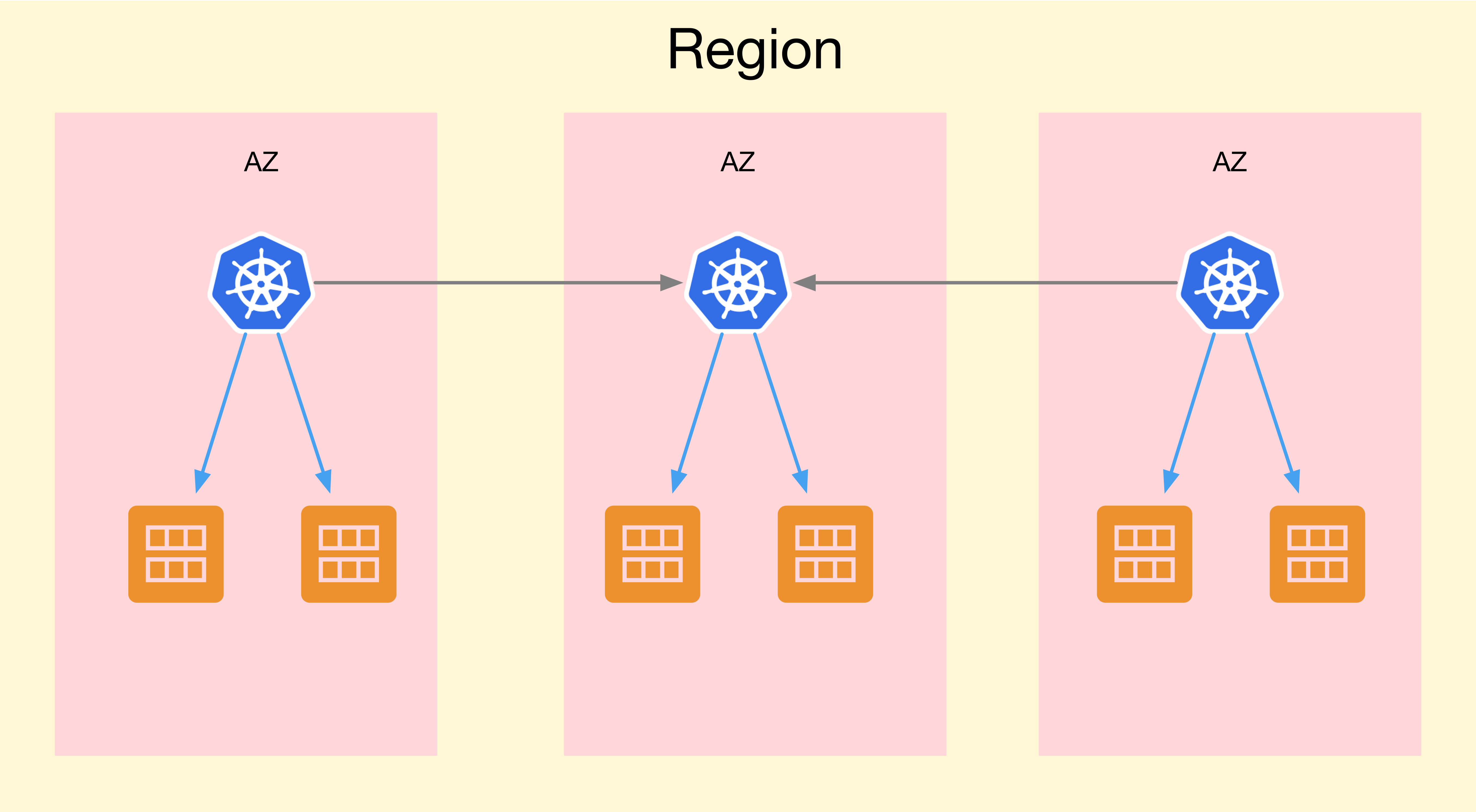
It's complicated, but one of the things I got tripped up by early on was all of the new terminology so let's dig deeper into that now

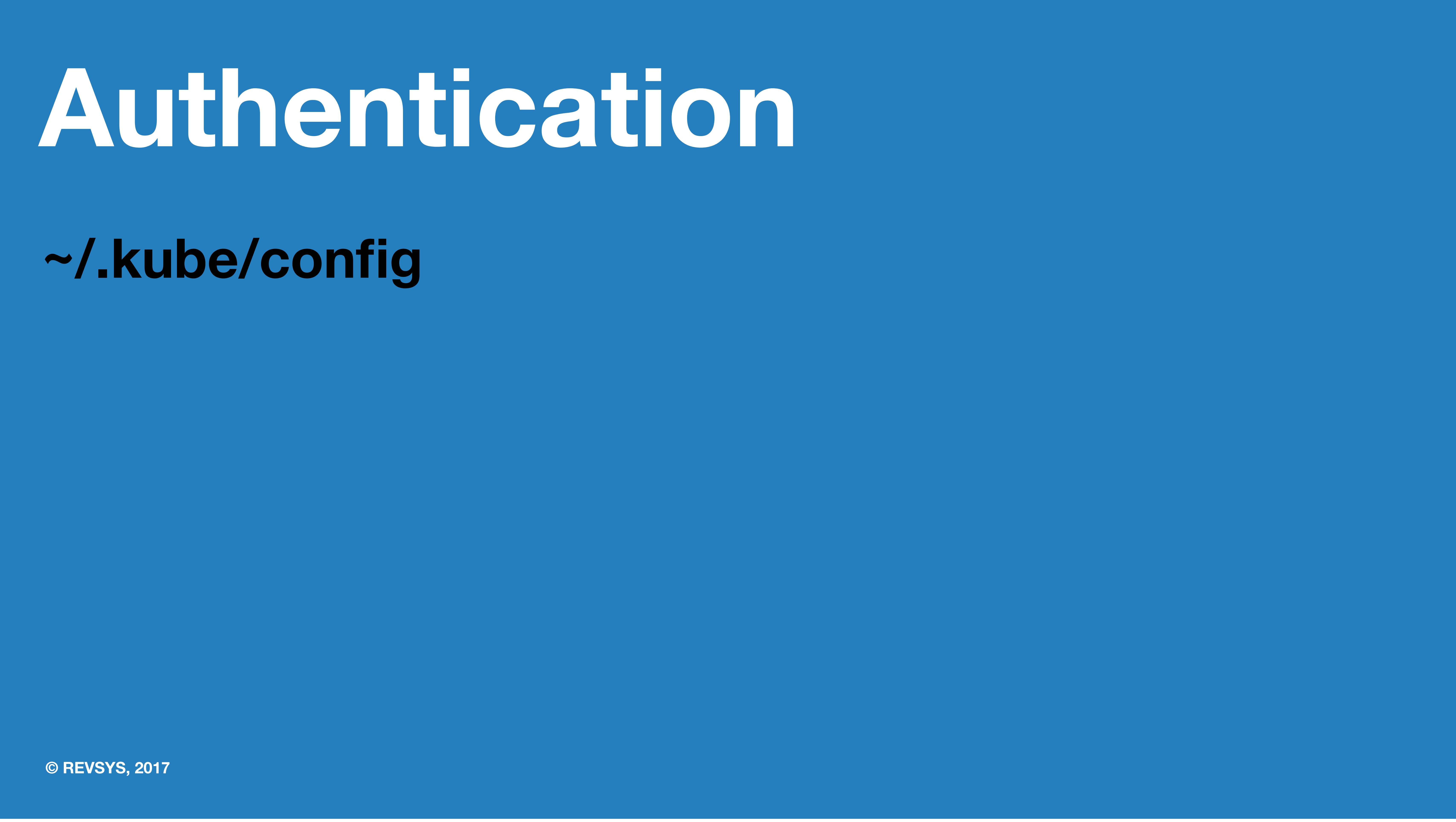
# k8s masters



# k8s nodes

© REVSYS, 2017


Region
AZ
AZ
AZ



# Authentication

**~/.kube/config**

Authentication, authorization, and access to the cluster can be as easy or as
complicated as you want to make it. Today we're going with the easy way,
so every operator of your cluster will just share a single .kubeconfig file.

# Access your cluster by proxy

```
$ kubectl proxy
```

**Now http://localhost:8001/ is proxied to your cluster's API**

# Dashboard

```
$ kubectl proxy
```

**Travel to http://localhost:8001/ui**

# namespace.yaml

```yaml
apiVersion: v1
kind: Namespace
metadata:
    name: revsys-rocks
```

**To create it in the cluster**

```
kubectl apply -f namespace.yaml
```

# Deployments

# deployment.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: revsys-rocks
  namespace: revsys-rocks
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: revsys-rocks
    spec:
      containers:
      - image: gcr.io/revsys-150116/revsys-rocks
        name: revsys-rocks
        ports:
        - containerPort: 80
```

# To create it in the cluster

```
kubectl apply -f deployment.yaml
```

# Services

# service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: revsys-rocks
  namespace: revsys-rocks
spec:
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: revsys-rocks
```

# Ingress Controllers

Ingress Controllers map the outside world into our services running in our cluster

# kube-lego

## https://github.com/jetstack/kube-lego

A quick aside, I'm about to show you an ingress controller that uses kube-lego, so I should explain what it is. kube-lego is a controller that handles Let's Encrypt certificates for us.

# ingress-tls.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: revsys-rocks
  namespace: revsys-rocks
  annotations:
    kubernetes.io/tls-acme: "true"
    kubernetes.io/ingress.class: "nginx"
spec:
  tls:
  - hosts:
    - revsys.rocks
    secretName: revsys-rocks-tls
  rules:
  - host: revsys.rocks
    http:
      paths:
      - path: /
        backend:
          serviceName: revsys-rocks
          servicePort: 80
```

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: revsys-rocks
  namespace: revsys-rocks
  annotations:
    kubernetes.io/tls-acme: "true"
    kubernetes.io/ingress.class: "nginx"
spec:
  tls:
  - hosts:
    - revsys.rocks
    secretName: revsys-rocks-tls
  rules:
  - host: revsys.rocks
    http:
      paths:
      - path: /
        backend:
          serviceName: revsys-rocks
          servicePort: 80
```

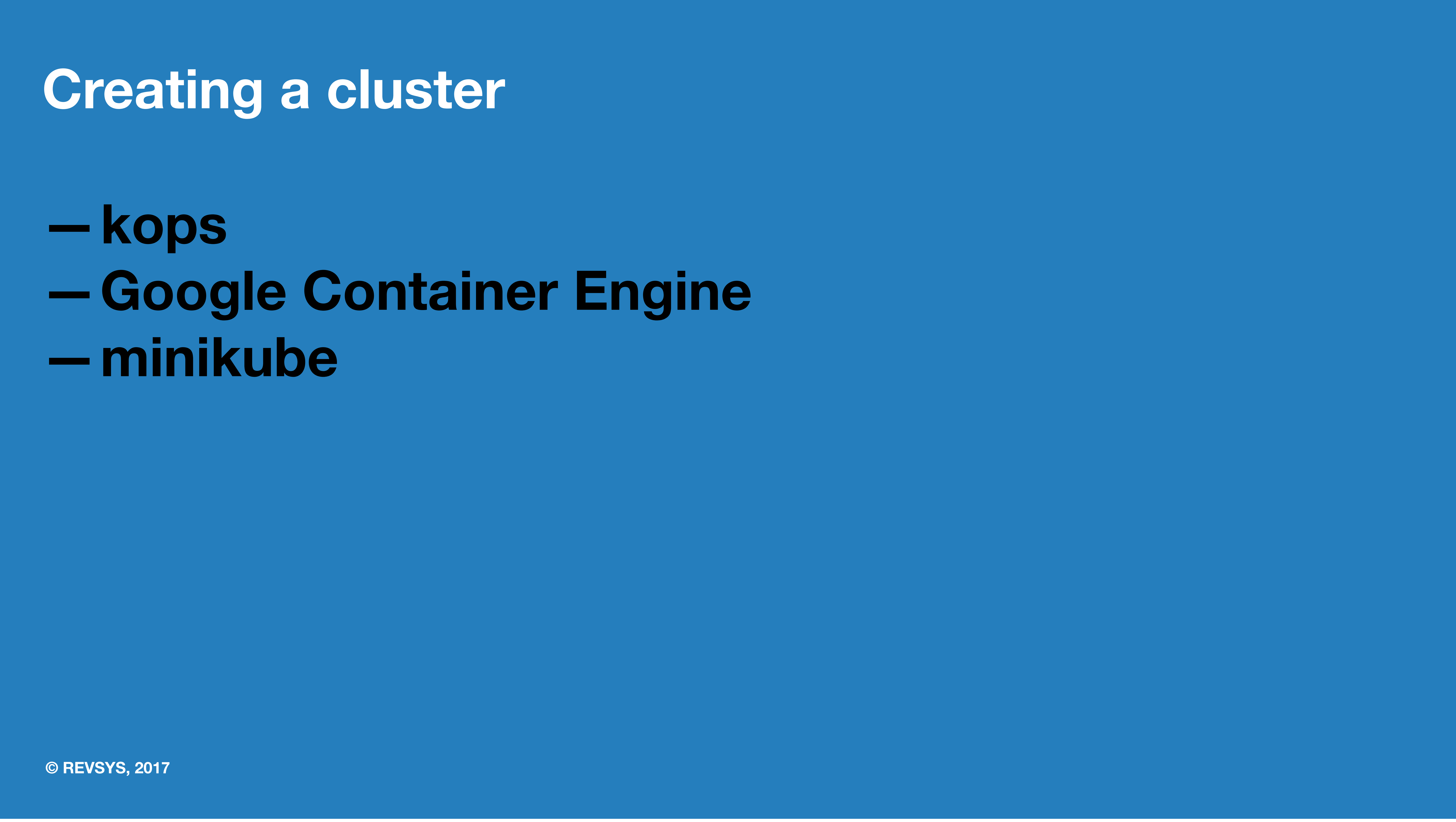
# Pods

When we create deployments, all of the containers in a deployment form a pod. Pods are sets of containers that are deployed together on the same host. This can be useful for many scenarios when containers need to work closely together. However, in my examples today all of our deployments have a single container. But you need to know about the concept to really understand the docs and various tutorials you'll find on Kubernetes

## So the high level view is...

1. The masters run the API, store cluster state, and schedule deployments onto the nodes
2. Nodes run pods which provide services inside the cluster
3. Ingress controllers map the external world to internal services

# Running kube in the real world



## Creating a cluster

- kops
- Google Container Engine
- minikube

There is also kubespray, an alternative to kops, which a few friends have mentioned using, but we haven't used it ourselves.

# Configuration

There are several different ways to pass in configuration information into your containers

# Environment Variables

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: revsys
  namespace: revsys-website
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: revsys
    spec:
      containers:
      - image: registry.revsys.com/revsys:v1.3.8
        imagePullPolicy: Always
        name: revsys
        env:
        - name: DATABASE_NAME
          value: "revsys.com"
        - name: DJANGO_SETTINGS_MODULE
          value: "revsys.settings.dev"
        ports:
        - containerPort: 80
      imagePullSecrets:
        - name: registry.revsys.com
```

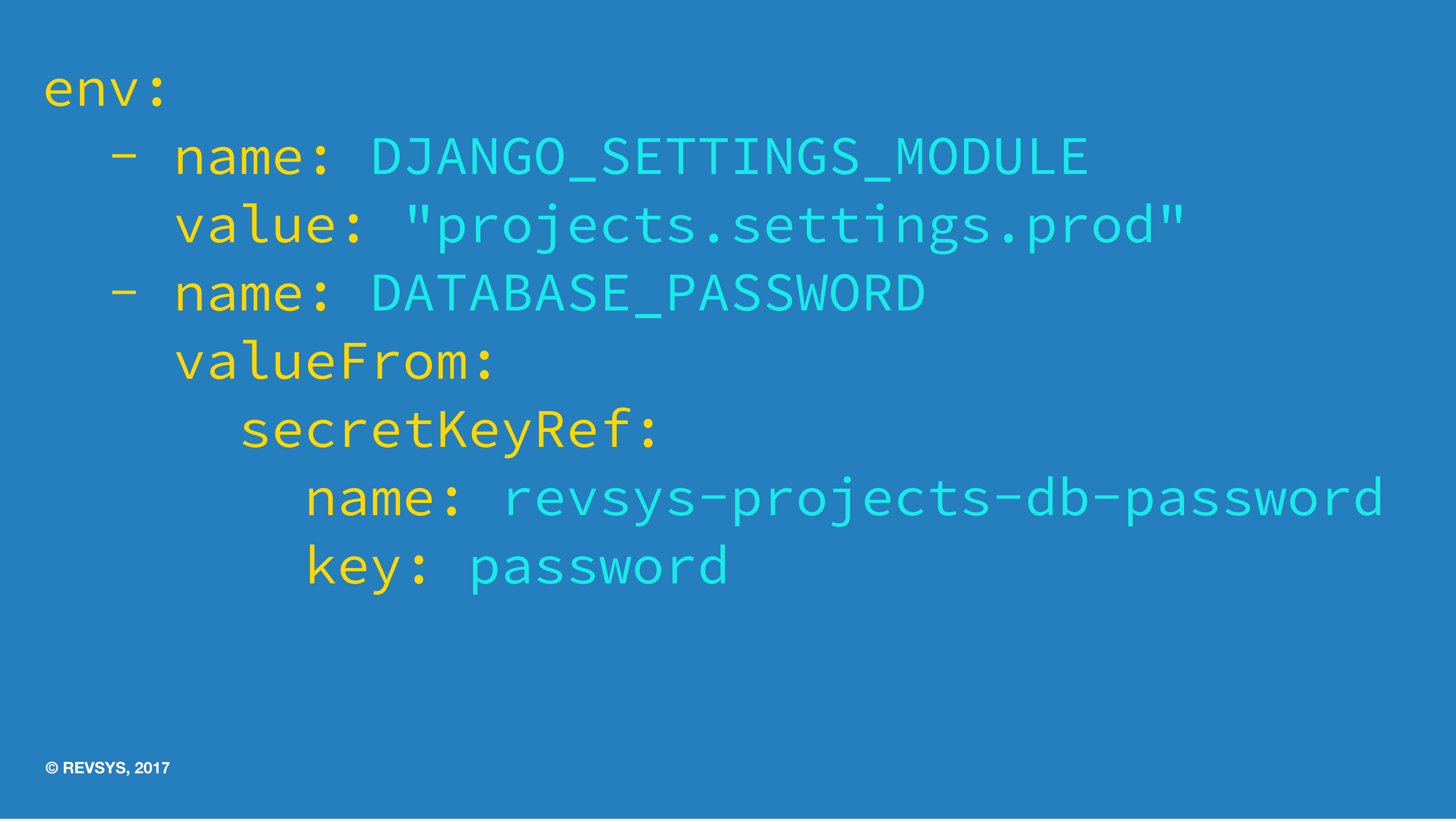
```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: revsys
  namespace: revsys-website
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: revsys
    spec:
      containers:
      - image: registry.revsys.com/revsys:v1.3.8
        imagePullPolicy: Always
        name: revsys
        env:
        - name: DATABASE_NAME
          value: "revsys.com"
        - name: DJANGO_SETTINGS_MODULE
          value: "revsys.settings.dev"
        ports:
        - containerPort: 80
      imagePullSecrets:
        - name: registry.revsys.com
```

# ConfigMaps

Config maps allow us to map sets of "variable" like things, whole files, or entire directories of configuration information into our Pods. Examples here don't map all that well to slides, but check out the documentation for more information. You can read up on them in the docs, but you can do things like a nginx config file and have it presented as an actual file in your container.

# Secrets

Kubernetes supports creating, defining, and managing "secrets". The obvious examples are API keys and database passwords. We could put these values in as environment variables directly, but that exposes are super secret information to more people than necessary. Kube let's us pull a secret in as an environment variable however so...



```
env:
  - name: DJANGO_SETTINGS_MODULE
    value: "projects.settings.prod"
  - name: DATABASE_PASSWORD
    valueFrom:
      secretKeyRef:
        name: revsys-projects-db-password
        key: password
```

So this is how we can easily use secrets, but how secret are they? Well right now they aren't that secure. They're stored as base64 on the cluster, but kubernetes is moving to support truly encrypted secrets in the next release.

# Centralized Logs are a Must

Since we never really know where things are running we absolutely need centralized logging. We have had luck with the EFK stack, specifically ElasticSearch, Fluentd or fluent-bit, and Kibana for this. Google's GCE automatically gathers up your container logs and makes them searchable for you.

# jslog4kube

## https://github.com/revsys/jslog4kube

# Data Persistence

# Persistent Volumes

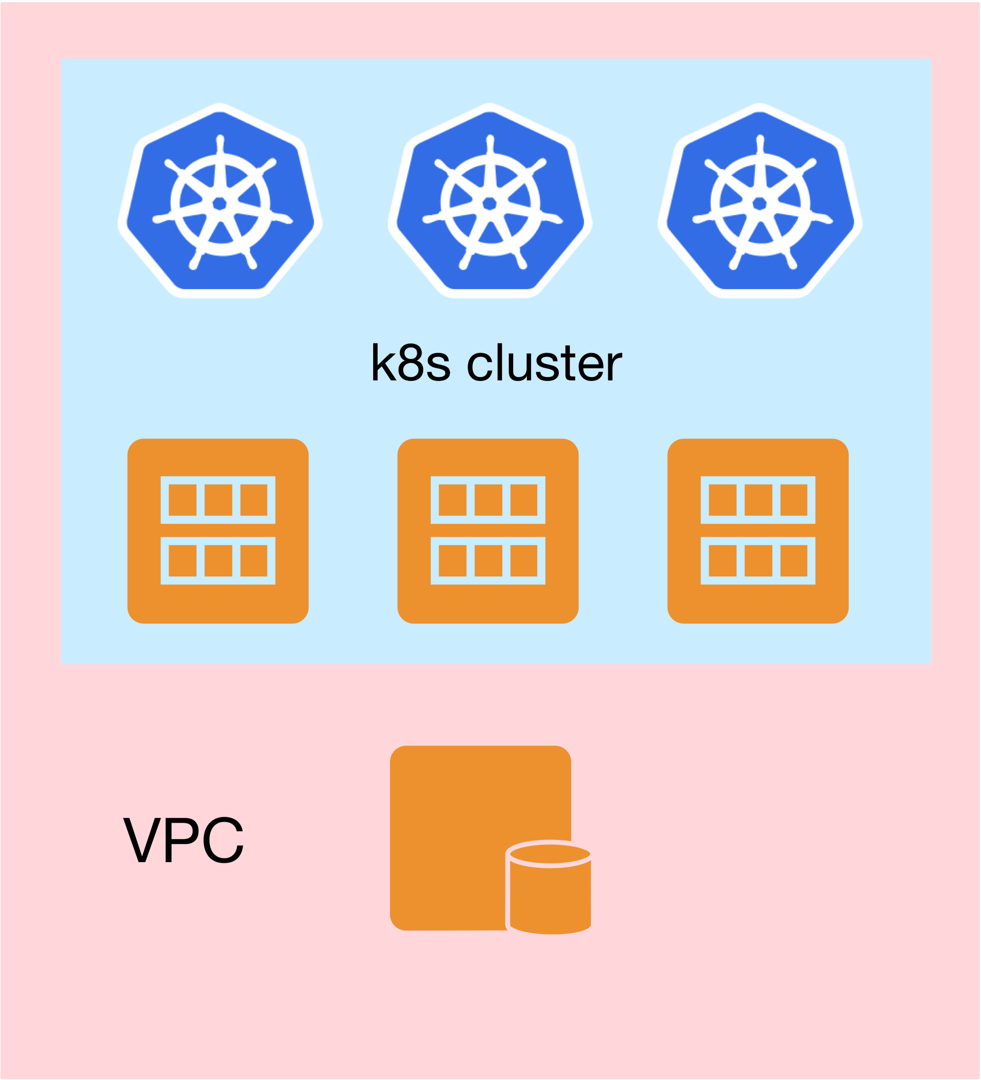
# Off Cluster Storage

# What about PostgreSQL?

## Take a look at Patroni.

## https://github.com/zalando/patroni

Patroni is a system for doing templated HA deployments of Postgres using something like Zookeeper, Consul or etcd. We haven't used it ourselves, but have heard good things so it's definitely worth investigating.



k8s masters

k8s nodes

persistent data instance

# helm

# Package Management for Kubernetes

https://github.com/kubernetes/helm

# Using the API with Python

```python
from kubernetes import client, config


config.load_kube_config()

v1 = client.CoreV1Api()
print("Listing pods with their IPs:")
ret = v1.list_pod_for_all_namespaces(watch=False)
for i in ret.items:
    print("{}\t{}\t{}".format(
        i.status.pod_ip,
        i.metadata.namespace,
        i.metadata.name,
    ))
```

# Output

```
(kube-demo) [kube-demo frank]$ python all-pods.py
Listing pods with their IPs:
10.0.0.3    grove-static    grove-static-1870186613-3bzr8
10.0.1.4    grove-static    grove-static-1870186613-gh2gm
10.0.2.150  hqcc    hqcc-3762067920-spm6t
10.0.1.10   hqcc    varnish-3334876750-38mdh
10.0.1.14   kssp    kssp-166829002-pjn14
10.0.2.145  kssp    varnish-3334876750-r6m33
10.0.1.3    kube-lego   kube-lego-3323932148-jpbf0
10.0.0.60   kube-system fluentd-gcp-v2.0-dzv4s
10.0.2.161  kube-system fluentd-gcp-v2.0-h5dc2
10.0.1.21   kube-system fluentd-gcp-v2.0-wn8k9
10.0.0.6    kube-system heapster-v1.3.0-1288166888-dq8v7
10.0.2.160  kube-system kube-dns-3664836949-78xcq
10.0.1.20   kube-system kube-dns-autoscaler-2667913178-39qxd
10.128.0.3  kube-system kube-proxy-gke-revsys-production-default-pool-4839b693-0dmc
10.128.0.4  kube-system kube-proxy-gke-revsys-production-default-pool-4839b693-kagb
10.128.0.2  kube-system kube-proxy-gke-revsys-production-default-pool-4839b693-u53t
10.0.0.4    kube-system kubernetes-dashboard-2917854236-2pjnm
10.0.0.9    kube-system l7-default-backend-1044750973-1fkjm
10.0.2.84   mentor-match    mentor-match-960480193-7s0fn
10.0.1.5    nginx-ingress   default-http-backend-3981334675-ptpz8
10.0.0.8    nginx-ingress   nginx-3757477279-z23xd
...
```

# Create your own operators

Why would you want to create your own controllers? Well kubernetes is great and flexible, but it doesn't handle everything you need all of the time. Using your own annotations and a bit of code to watch for them you can take actions inside and outside of the cluster when things change or need to change.

# Examples of operators you could build

1. Slack alerts when new deployments are created or when pods come up and down
2. Watch for your Django apps and automatically back up all databases in use
3. Orchestrate more complicated scenarios that k8s doesn't support directly, for example swapping out a service for another after a long running setup period.

# Questions?

## Twitter: @fwiles

## Email: frank@revsys.com