Getting Started With Kubernetes

AIM | hUc

Hello!

Today I'm going to try to cover as much of k8s as I can in the time we have. It will, by nature be a shallow dive, but please see me afterwards, I would *love* to talk more about it.

My name is John Hobbs, you can find me as @jmhobbs on Twitter or Github.
I am a software engineer at Flywheel here in Omaha, so I get to work on the best WordPress hosting in the world with these amazing people.

So, first things first, what is k8s?
The subtitle on the website is “Production grade container orchestration.”
You’ve got some applications you need to run, and some servers you can run them on, and Kubernetes takes care of the rest of it.

K8s comes out of Google and the Borg system that's been in use there for years.

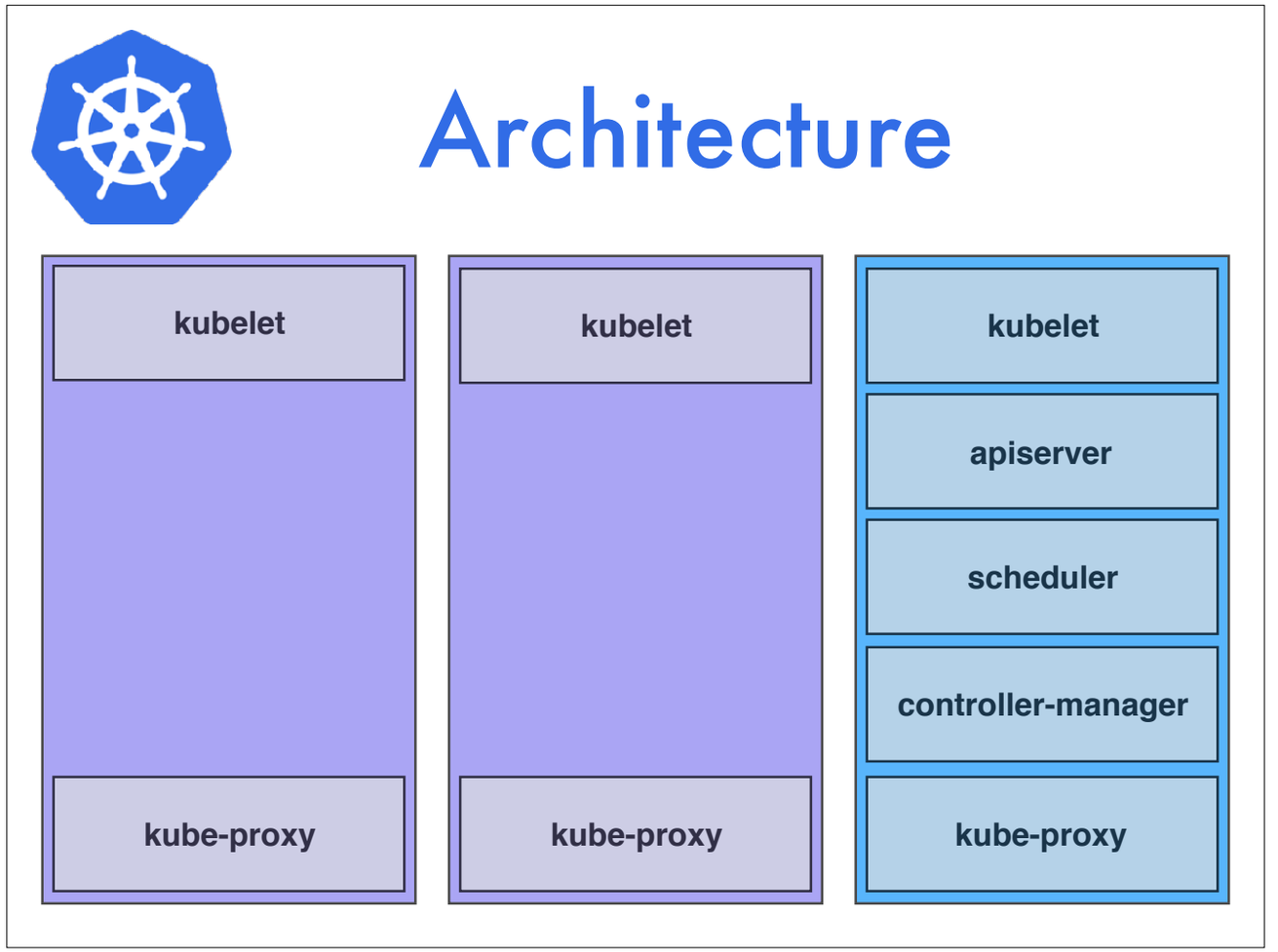
https://research.google.com/pubs/pub43438.html

Kubernetes is...

- Open source
- Moving fast, but very stable
- Container based
- Self-healing*
- Pretty easy to use

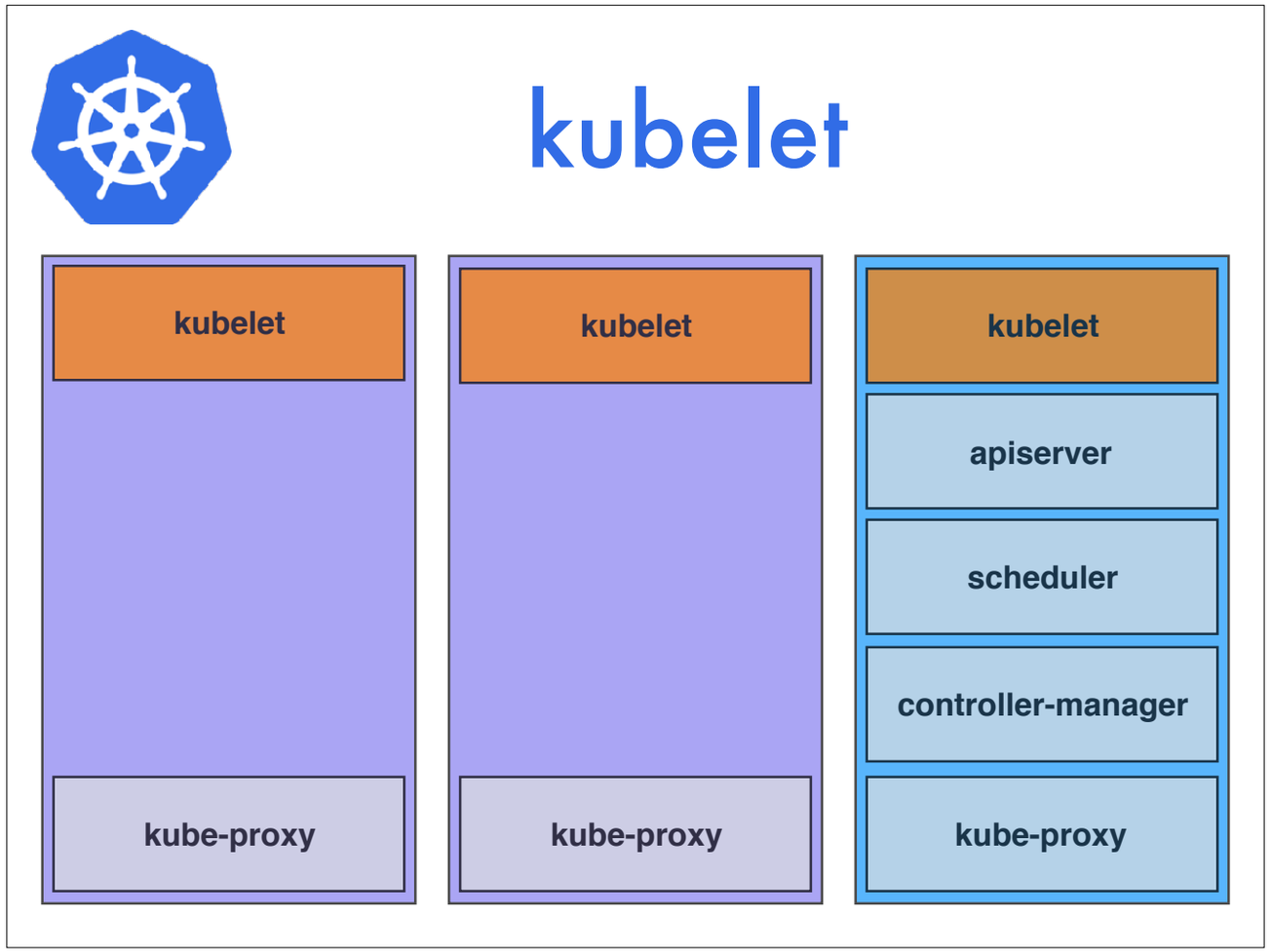
* Though led by Google, development is open and external contributions are high.
* New version every 3 months, backwards compatible unless using alpha/beta API's
* By default, k8s uses Docker, but it can use rkt as well
* If you run k8s across multiple servers (you should) it will move containers around if a server fails
* To get started, you only need to know a few basic things

Let’s pause quick for a reality check. Kubernetes is very cool, and very hot right now. That said, it’s a large conceptual undertaking for an organization. This will not get rid of all your operators, but it can relieve some of their workload and make your team more agile. Fair warning.



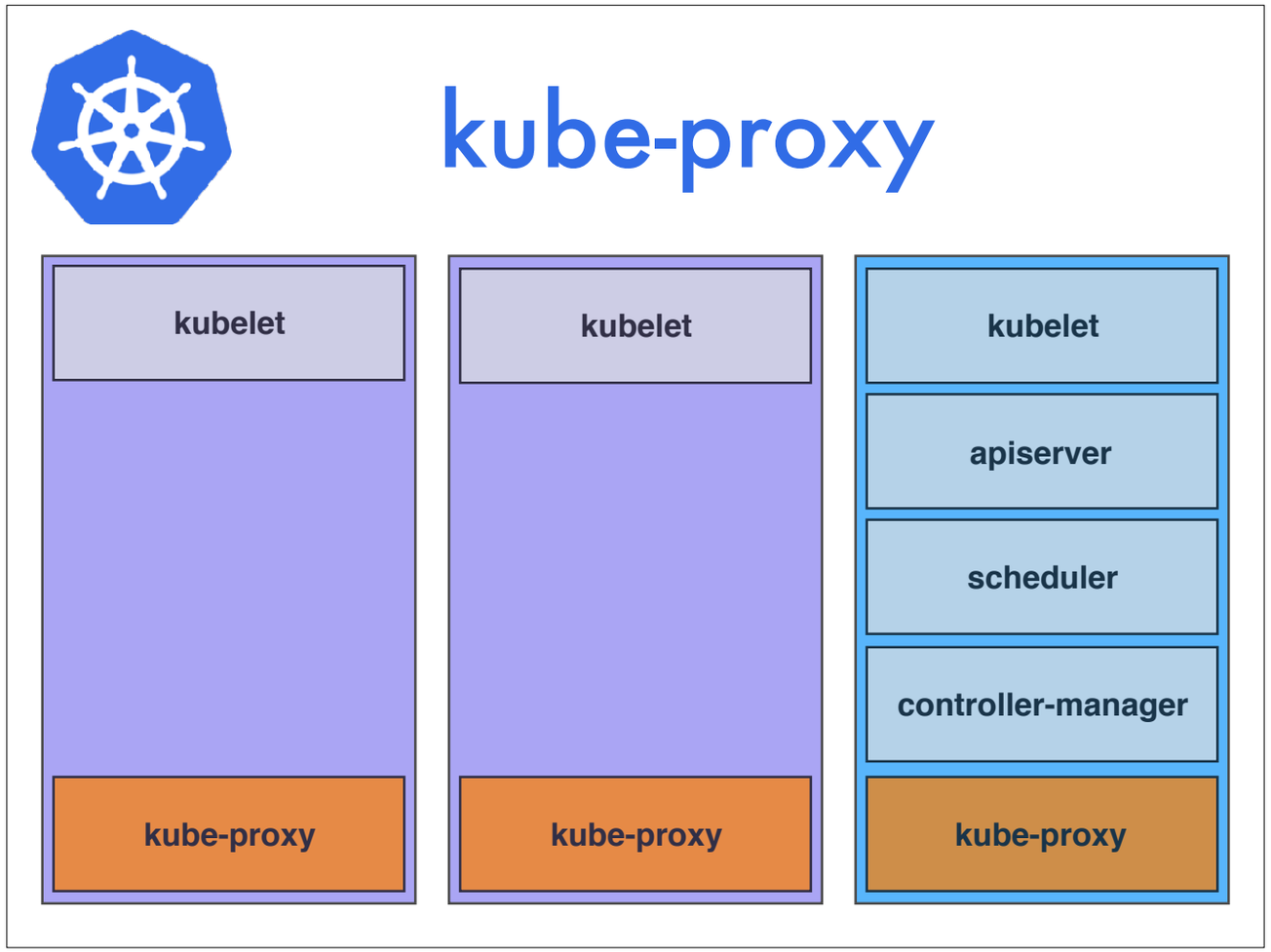
So this is a simplified overview of the architecture of Kubernetes. The purple boxes are worker nodes, these are the servers where your applications will be run. The blue box is a master node, which does not get work scheduled to it by default, and runs the software that coordinates the cluster and distributes work.

In a typical installation you will have redundant master nodes which provide failover and extra capacity. Worker node count can be as low as one or as high as 5000.



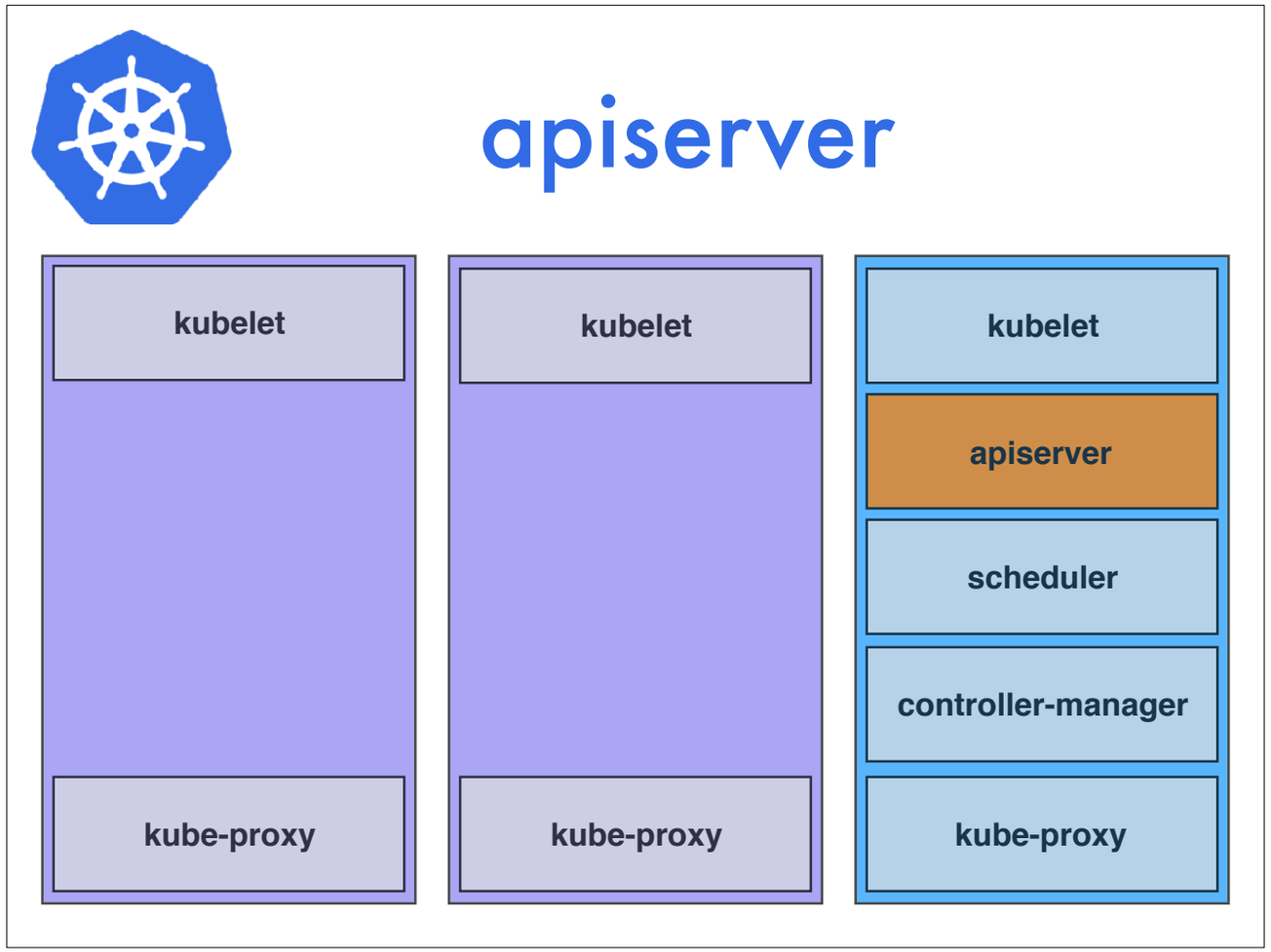
The most fundamental component of k8s is the Kubelet. This runs on every node in the cluster and has one main job. It takes manifests of containers you want to run on that node, and it makes sure they get run. The kubelet can actually stand alone. One way to provide containers to run is via the filesystem, and the kubelet will happily manage those for you without being part of a k8s cluster.

https://kubernetes.io/docs/admin/kubelet/



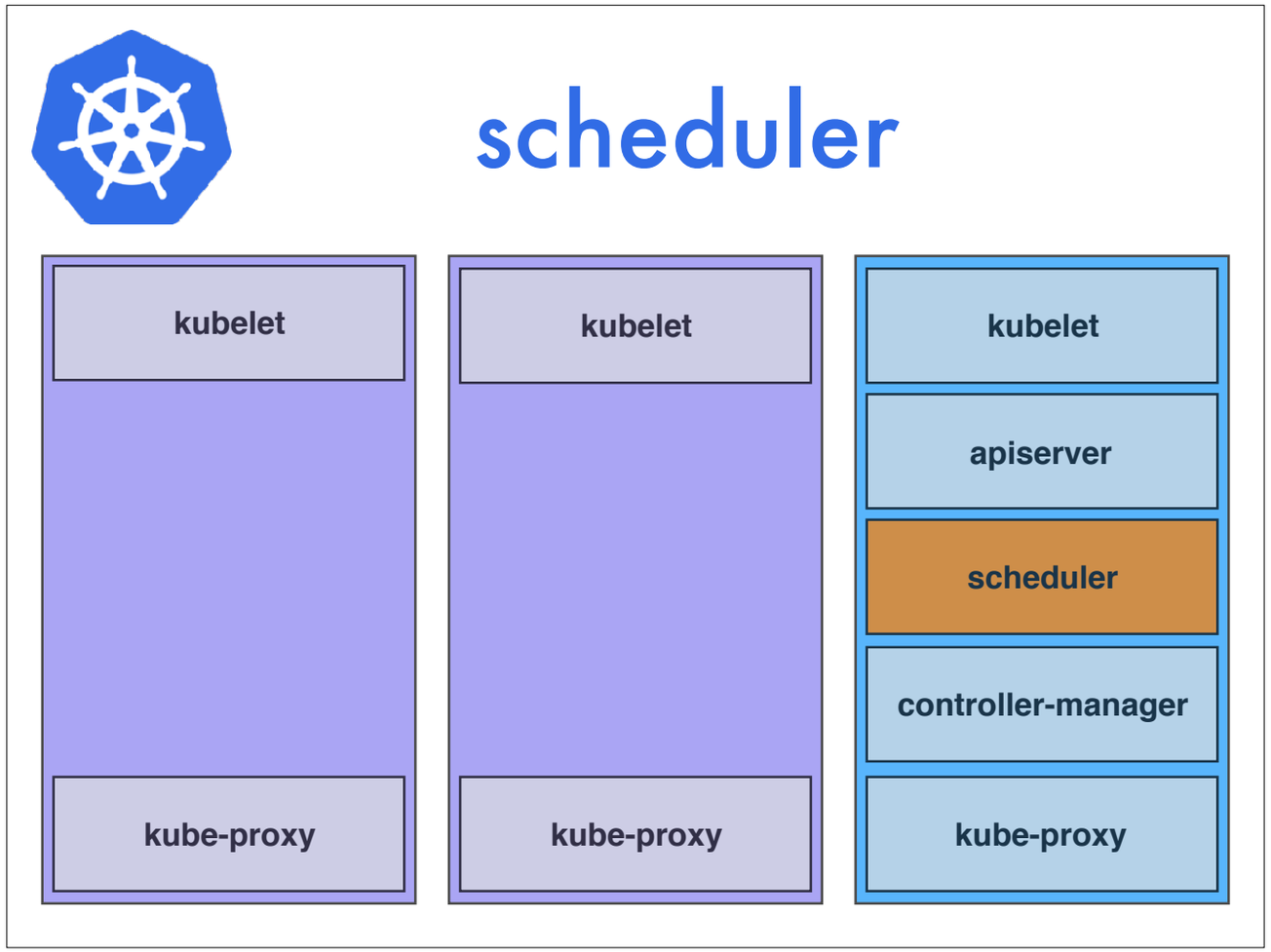
The kube-proxy is a service that runs on every node to provide a proxy for all of the services running in the cluster. This way, you can get routed to a node which can serve the service you're looking for without doing any special mapping in your application. This should become more clear later on.

https://kubernetes.io/docs/admin/kube-proxy/



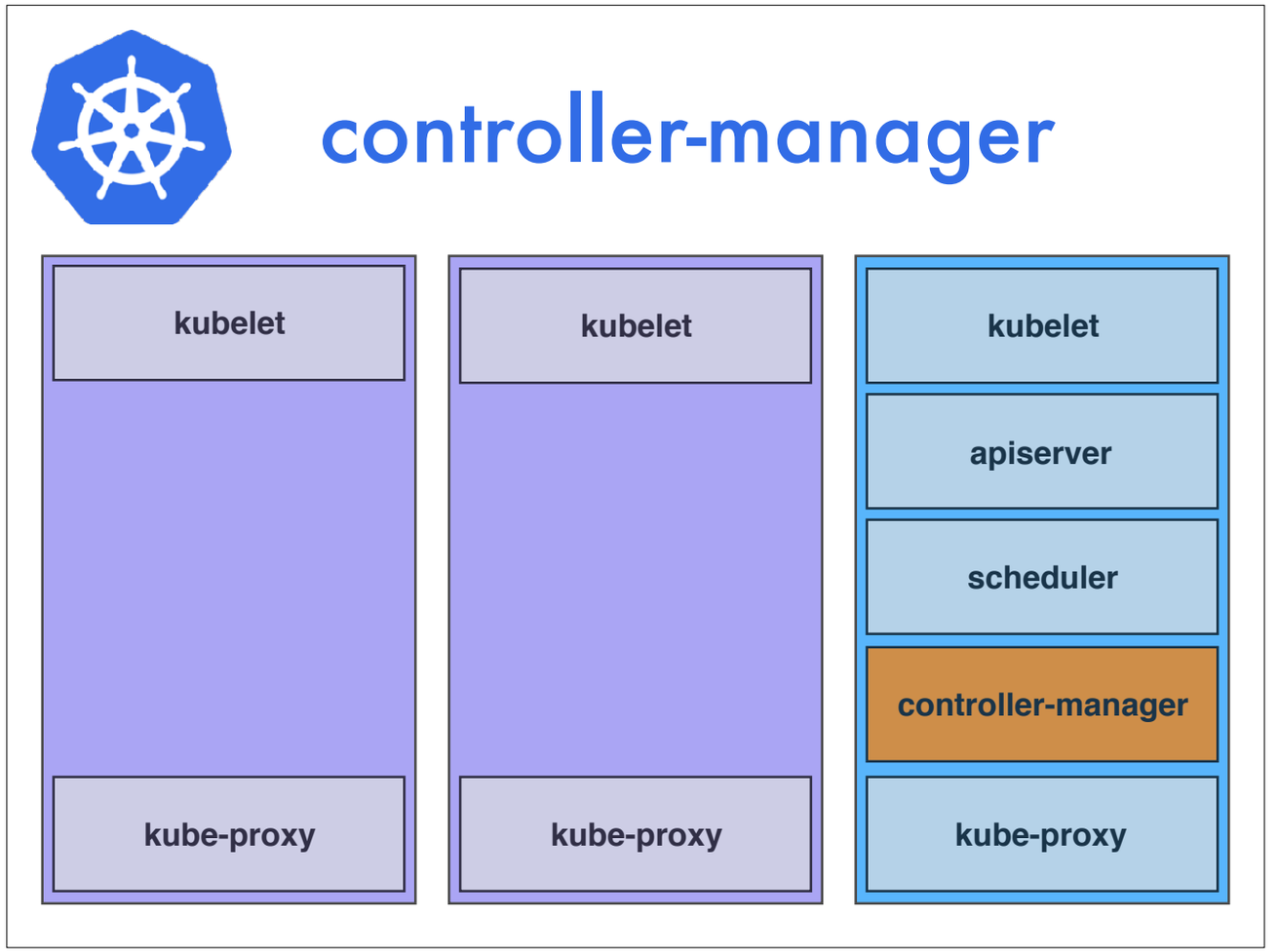
The API Server is the core of the k8s system. It handles validation of all the objects in the k8s cluster and acts as the frontend to the shared state, which is stored in etcd. If you want to inspect or alter Kubernetes managed objects, you do so through the apiserver.

https://kubernetes.io/docs/admin/kube-apiserver/



The scheduler is what it says on the tin. It takes workload requirements and try to find an available node to run it on. This is more complicated than it sounds though, as there are lots of constraints you can place on your workloads that make scheduling more difficult.

https://kubernetes.io/docs/admin/kube-scheduler/

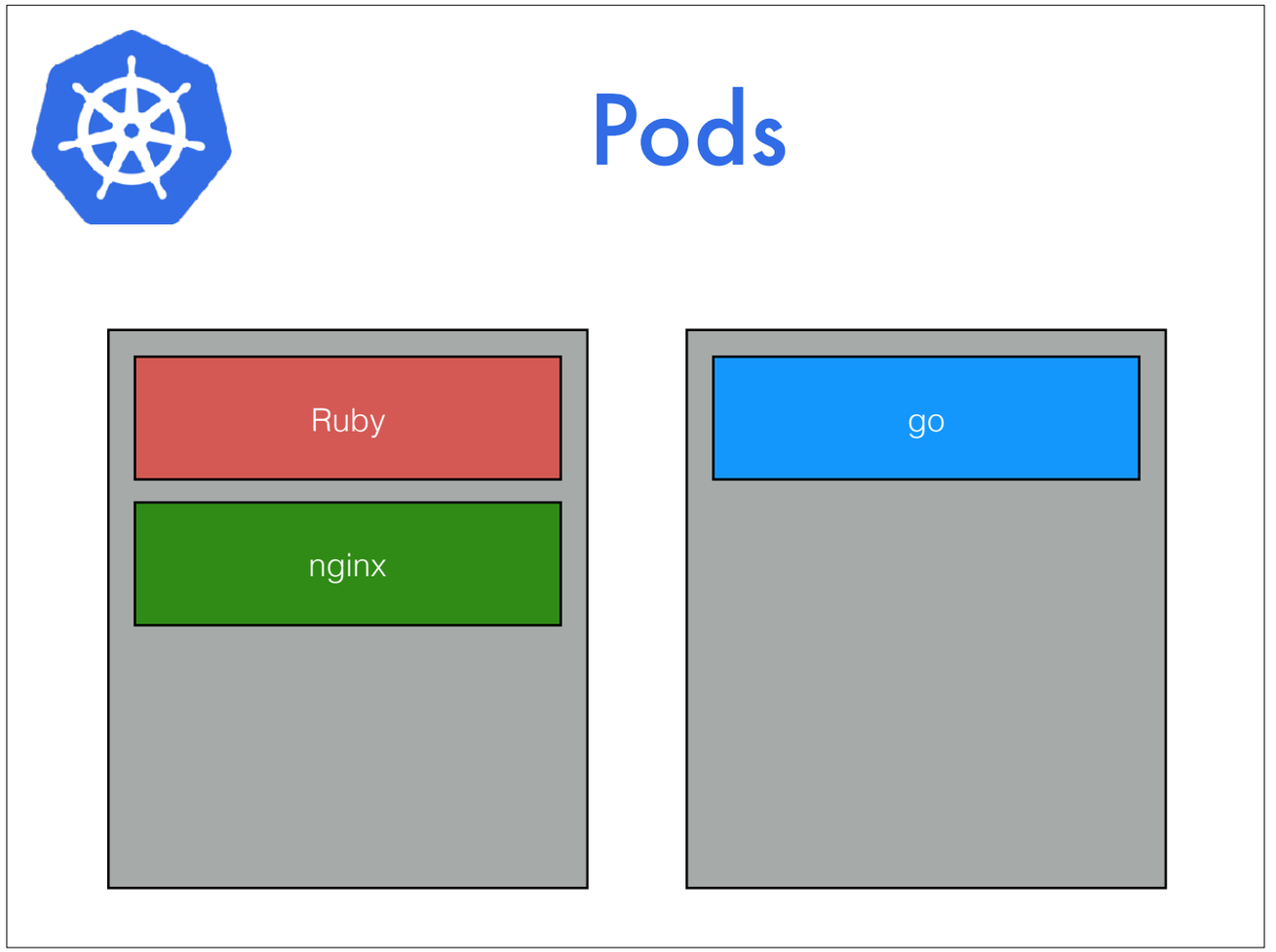


In k8s, a Controller is a loop which watches the state of the cluster through the apiserver and makes changes in an effort to move the current state towards the desired state. Examples being the service controller, node controller, etc. This is the engine of Kubernetes orchestration.

Several other components can be in a Kubernetes cluster, but they aren't required.

* The two most common are DNS via SkyDNS and an overlay network, most often via Flannel. Since 1.3 DNS is baked in.
* Logs are pretty important. What you use depends largely on provider, but Fluentd and the ELK stack are common.
* The kubernetes dashboard is a great add-on. It gives you a web UI to view and edit the state of your cluster.
* Heapster aggregates metrics from the nodes. Prometheus is probably the leader in monitoring and alerts right now.
* You can declare network policies in k8s, but you'll need an add-on like Calico to enforce them.
* Autoscalers can increase the number of nodes in your cluster, or the number of containers for a service.

Okay! Now on to the practical stuff.



The most foundational object in k8s is the Pod. A Pod is a logical wrapper around one or more Docker containers. It is the indivisible unit of work.

Here's a pod with a ruby container, say a Sinatra app. And there's also an nginx container that's doing gzip and serving static assets in there. These two containers will always be running on the same server, they share storage and network resources.

Here's another pod with just a single container running a go app. It's just one container in the pod, that's ok too.

https://kubernetes.io/docs/concepts/workloads/pods/pod/

Pods

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: sinatra
    image: jmhobbs/my-sinatra-app:latest
  - name: nginx
    image: nginx:latest
```

Everything in Kubernetes is declarative. You specify the state you want to exist, and k8s does the work to make that happen. As such, everything adheres to a well defined specification. Most often, you will inspect and edit that in YAML.

Here is a minimal example of the Pod specification from the last slide.

Pod Resources

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: sinatra
    image: jmhobbs/cats-for-gold:latest
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

By default, containers can use as much CPU and memory as they please. It would be nice if we lived in a world without limitations, but we do not. Hence, Kubernetes has resource requests and limits.

A request is a minimum reserved amount of memory and CPU that a container requires to run.
A limit is the maximum amount of memory or CPU a container should be allowed to consume.

A container that exceeds it's memory request, it could get rescheduled when the node runs out of memory. A container that exceeds it's memory limit may be terminated.

A container that exceeds it's CPU request doesn't have any major repercussions. One that exceeds it's limit may be throttled.

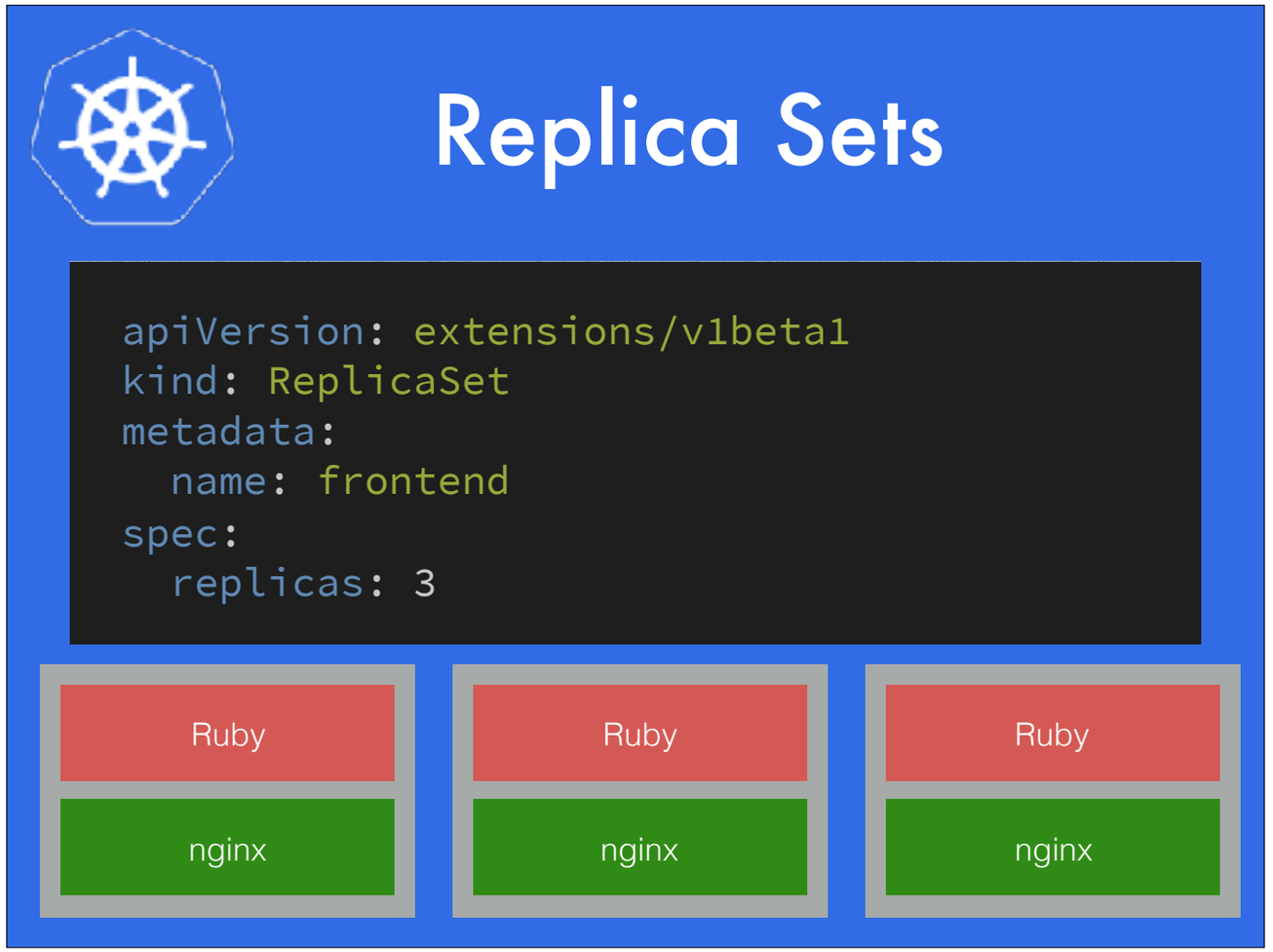
https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

Our second object is the replica set. A replica set keeps a specific number of copies of your pod running.

Here's some YAML that partially defines a replica set. In addition to what you see here, there would be a nested pod specification as well as other elements.

If you ask for one replica, you get one pod.

https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/



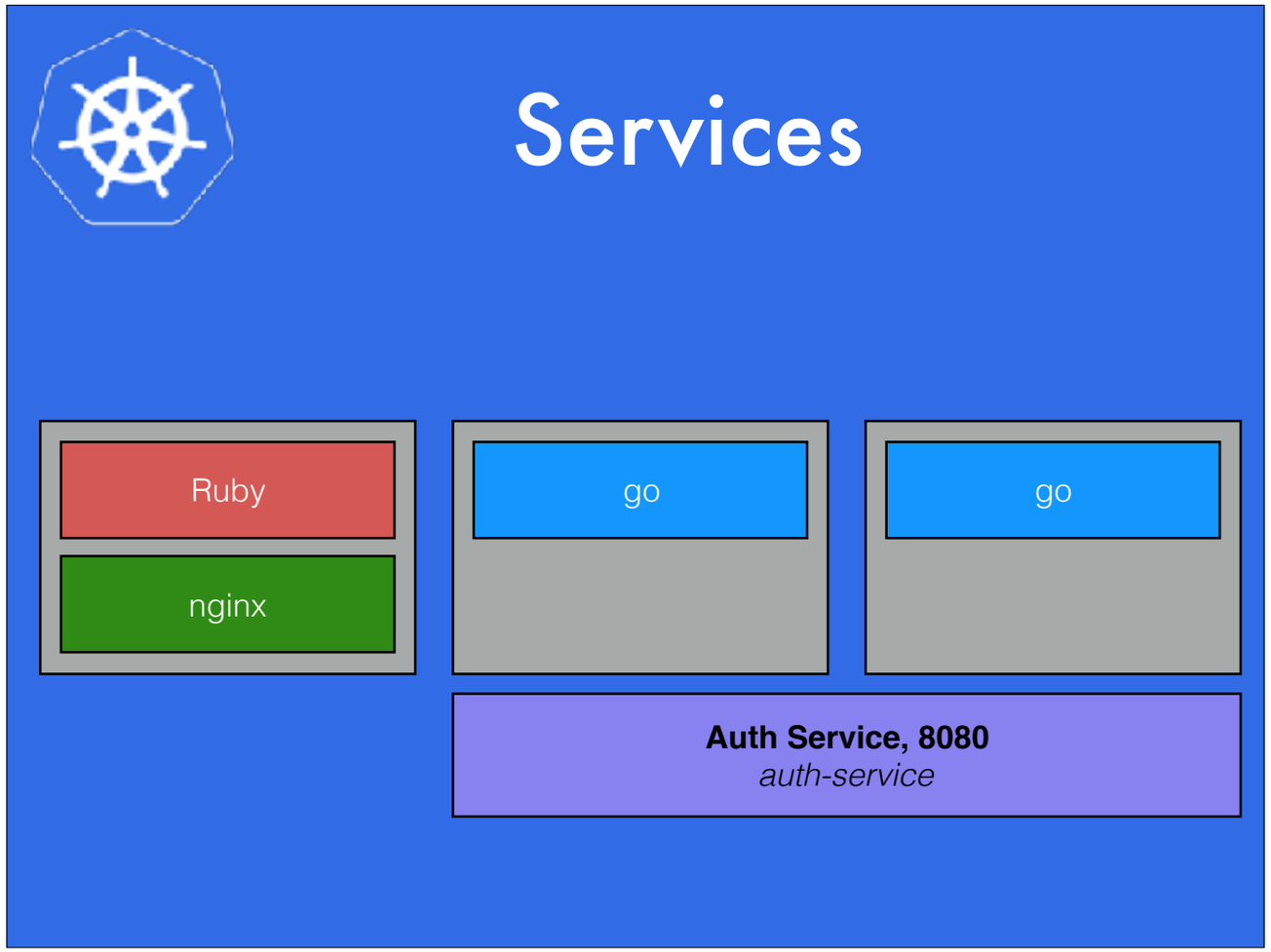
If you ask for three, it will try to run three.
If one pod gets destroyed or fails, the replica set will see that and attempt to create a replacement.

Deployments

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: cats-for-gold
spec:
  replicas: 2
  revisionHistoryLimit: 3
  template:
    spec:
      containers:
      - name: sinatra
        image: jmhobbs/cats-for-gold:v1
```

Building on top of replica sets is the deployment. A deployment keeps a history of replica sets and will manage rolling them forward and back as requested. This lets you do blue/green rolling deploys of your pods by scaling down one replica set and scaling a new one up in tandem.

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/



Ok, so we have all these pods running, how do we communicate between them? Services. A Service is a persistent way to address a set of Pods, regardless of when they are created or destroyed.

Let's say we have a pod for our web front end. We also have two pods for our authentication API, written in go. Our frontend needs to be able to talk to the authentication backend, but it can't rely on the IP's of the other pods, because they are ephemeral.

So, we create a service on port 8080. Now, we can access our auth service using the DNS name "auth-service"

https://kubernetes.io/docs/concepts/services-networking/service/

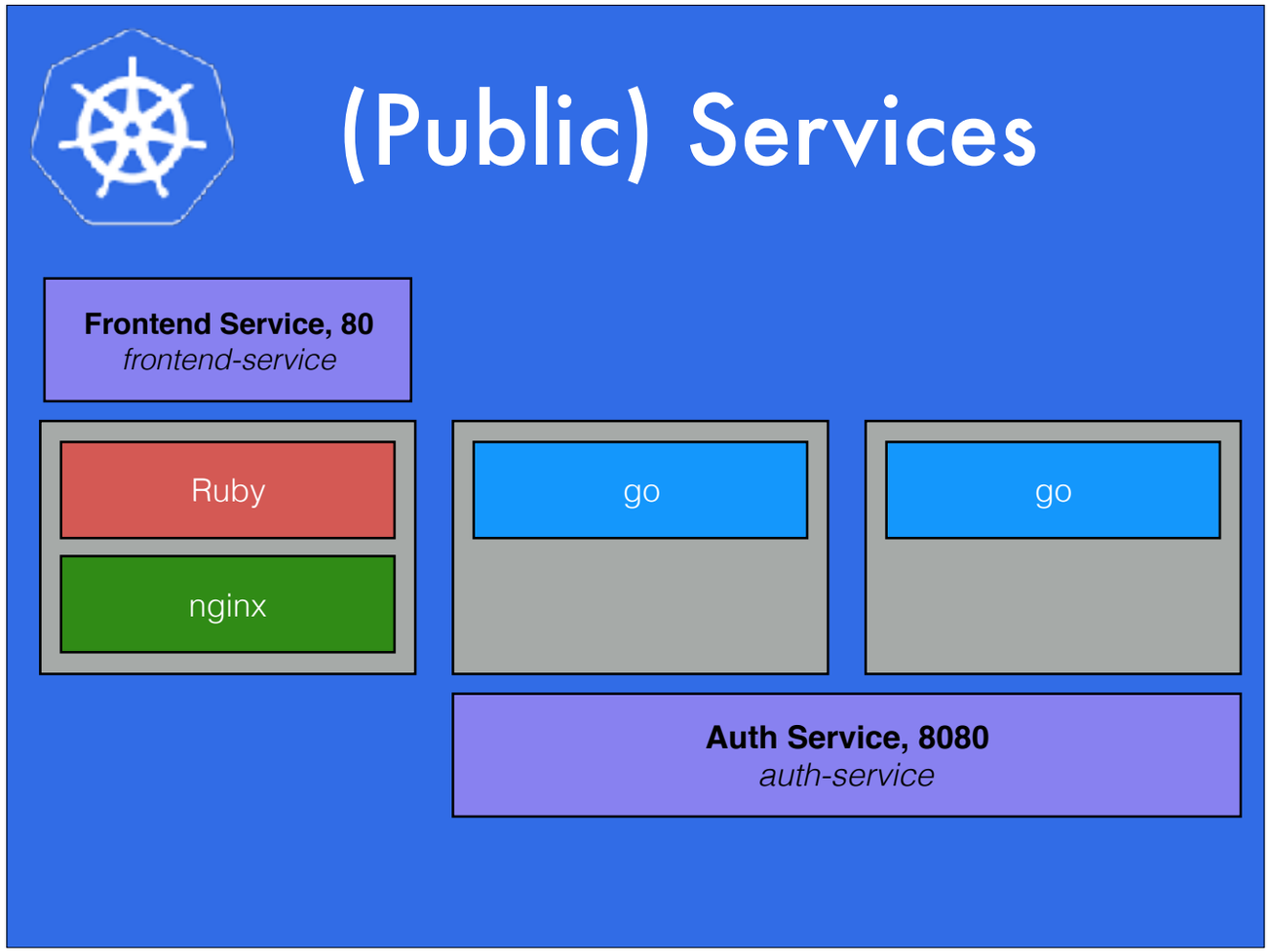
Services

```yaml
apiVersion: v1
kind: Service
metadata:
  name: auth-service
spec:
  ports:
  - port: 8080
    targetPort: 80
  selector:
    app: cats-for-gold
    component: authentication
```

This is a service definition. Note that the inbound port 8080 doesn't have to match the port in the pod, 80. Also note the selector section. I haven't discussed labels yet, but they become vital for services.

Services will roughly load balance between multiple pods as they become available. To determine which pods to address, we use label selectors. When defining a pod specification, you can set arbitrary labels. In this example I'm looking for pods with the value "cats-for-gold" for app, and "authentication" for component.

The services controller will watch for changes to those selectors and update the service endpoints as needed, ensuring traffic goes where it belongs.



There are several ways to expose your apps to the internet at large, but they all revolve around services. I'll discuss load balancers first. Kubernetes has support for various cloud platforms native load balancers, each with their own quirks. Google cloud is fairly easy to work with. You mark the service as type "LoadBalancer" and GCE will provision a TCP load balancer for us, then route traffic from that to our frontend pods.

https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services---service-types

LoadBalancer

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: cats-for-gold
    component: frontend
```

Here is a LoadBalancer service. It's very similar to the other service, except that it specifies it's type as LoadBalancer.

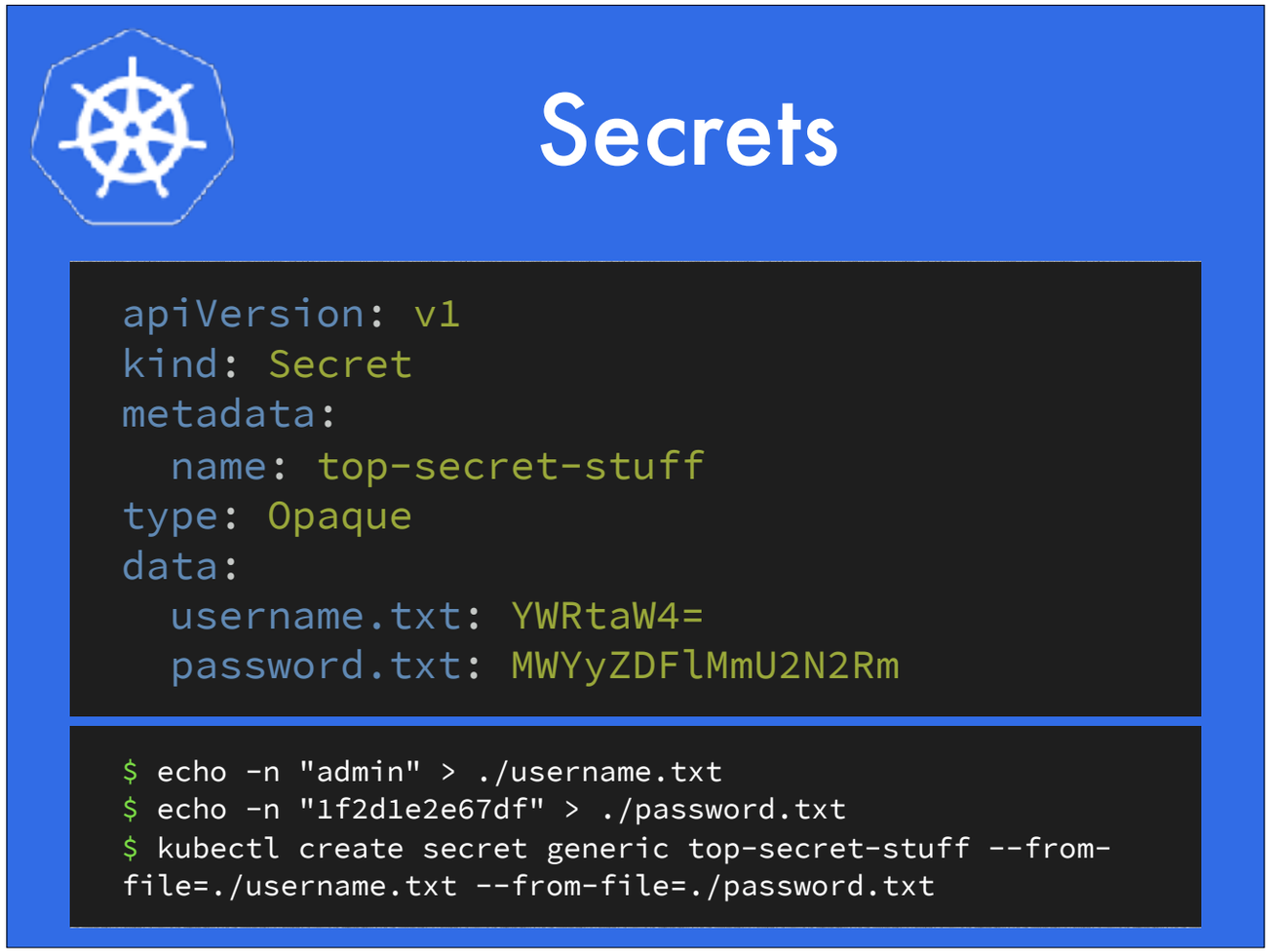
NodePort

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30001
  selector:
    app: cats-for-gold
    component: frontend
```

A NodePort service will reserve a high level port, 30k to about 32k by default, just for your service. You can also specify a port in this range if you would like, as I've done here. Any traffic that hits any node in the cluster on that port will be routed to your service by kube proxy. This is mainly useful for building your own load balancer infrastructure.

Demo!

Ok! Let’s take a quick break from slides and try a demo tying all of this together.



Let’s talk about a few more minor things before we get to the last big component.

A secret is a Kubernetes object that can be used to contain sensitive information. It’s safer than baking it into the Pod spec, and you can mount the contents as a volume. Creating secrets is kind of a pain, even with kubectl to help.

https://kubernetes.io/docs/concepts/configuration/secret/

Secrets

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-example
spec:
  containers:
    - name: secret-consumer
      image: jmhobbs/blabbermouth:latest
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: top-secret-stuff
              key: username.txt
```

One way to consume a secret is to expose it as an environment variable to a container, like so.

ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: doge-config
data:
  much: wow
  very: doge
  such: kubernetes
```

A ConfigMap is like a non-secret secret. Again, you have key/value pairs, and you can likewise consume them via environment variables.

https://kubernetes.io/docs/tasks/configure-pod-container/configmap/

I'm sure you've noticed that I have not yet broached the subject of storage yet. Volumes can solve many storage issues. Sharing data between containers in a pod, checkpointing to recover from crashes, holding content to be served, etc.

Volumes are contained by Pods, and their lifetime is bound to the lifetime of a Pod. Every container inside of a pod can mount the volume.

Kubernetes supports many, many volume types, each with their own configuration. I'll show a few and let you check the docs on the rest.

https://kubernetes.io/docs/concepts/storage/volumes/

Here is an emptyDir. You can see it's definition at the very bottom there. An emptydir is a temporary directory that is created on the node when a pod is scheduled to it. When a pod is destroyed or removed from a node, the directory is too.

Let's say I have a caching proxy but it likes to crash. A lot. I could make an emptyDir like this to store my cached files in so that the cache is already warm after a crash of my container.

Volumes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-example
spec:
  containers:
  - image: jmhobbs/doge:latest
    name: doge
    volumeMounts:
    - mountPath: /etc/config
      name: doge-config
  volumes:
  - name: doge-config
    configMap:
      name: doge-config
```

This one is an example of a ConfigMap mount. Whatever keys are defined in the doge-config map will be exposed as files in /etc/config. The really cool thing here is that if we then update a value in that config map, it will be updated in the mount! This way if you watch a config directory, your software can detect config changes and reload them on the fly. Very cool.

Volumes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-example
spec:
  containers:
  - image: jmhobbs/webserver:latest
    name: static-webserver
    volumeMounts:
    - mountPath: /var/www
      name: www-data
  volumes:
  - name: www-data
    gcePersistentDisk:
      pdName: my-website
      fsType: ext4
```

Sometimes you need storage that lasts. If you're on GCE, a Persistent Disk is ideal. These are not deleted after they are used like an emptyDir, so it can be passed around between pods with some coordination. These can also be mounted read only by multiple pods, which can be useful.

There are lots of other examples, but they follow the same general design.

Persistent Volumes

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-100-001
  spec:
    capacity:
      storage: 100Gi
    accessModes:
      - ReadWriteOnce
  gcePersistentDisk:
    pdName: gce-pv-100-001
    fsType: ext4
```

Volumes are bound to Pods and many are provider specific, but it often makes sense to abstract the implementation of storage from how it is consumed. This is where Persistent Volumes come in.

A persistent volume is a provisioned piece of storage, which is available in the cluster to be claimed. PV's can also be dynamically provisioned, but that is outside the scope of this talk.

When a user needs storage, instead of specifying how they want storage, they specify what they want for storage, and Kubernetes finds the resources they need from what is available.

This is an example of a GCE PD. It's 100 gigs in size, and can be mounted read write to one node only.

To use this PV, I would create a persistent volume claim.

https://kubernetes.io/docs/concepts/storage/persistent-volumes/

Persistent Volumes

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: example-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 50Gi
```

Note that in my claim, I am not specifying what backend I want for my storage, only how much I need and what access policy I want to use.

Also note that while my request is for 50Gi, I will match with the 100Gi PV. PVC's can bind to PV's that are bigger than requested, but never to ones that are smaller.

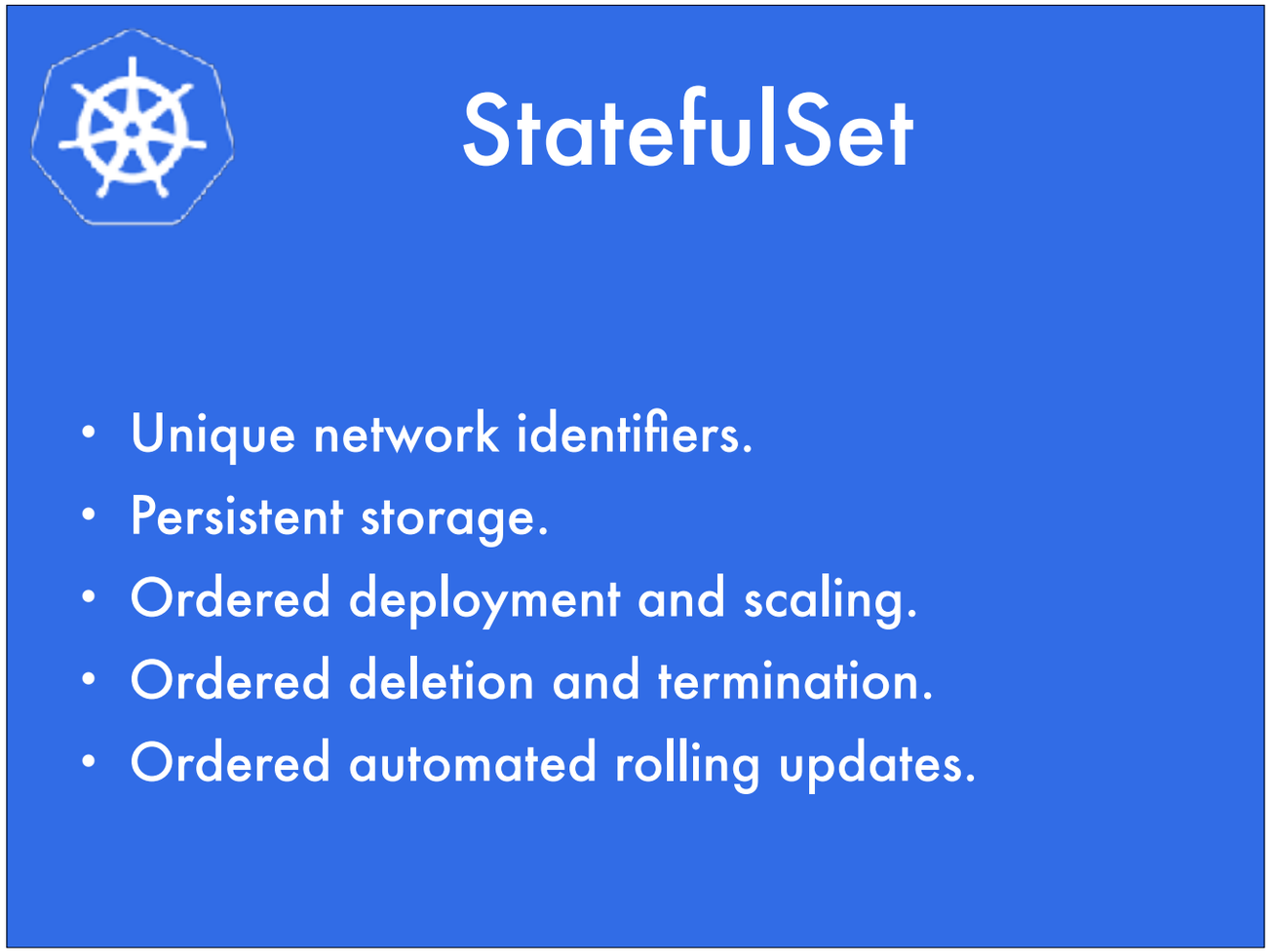
https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims

Persistent Volumes

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: cats-for-gold
spec:
  containers:
    - name: frontend
      image: jmhobbs/cats-for-gold:v1
      volumeMounts:
      - mountPath: "/var/www/html"
        name: pvc
  volumes:
    - name: pvc
      persistentVolumeClaim:
        claimName: example-pvc
```

At the Pod level, we consume PVC's quite simply as a volume type.

Ok, great. We've got pods, services and storage. What if I want to run something that's not container native and needs to be a special snowflake? Something like MySQL perhaps?



Enter StatefulSet! The specification of a statefulset is almost identical to a deployment, with a few caveats. I'm going to largely gloss over this because I've not utilized it personally.

Storage and network identifiers for a pods are consistent and persist across rescheduling of the pods to new nodes. When deploying, scaling or updating pods, each pod in the set will not start until it's predecessor is in a running state. This lets you set up leader/follower relationships easily. Likewise, termination will start with the highest cardinality replica and work down to zero.

https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

DaemonSet

```yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
spec:
  template:
    metadata:
      name: fluentd-elasticsearch
    spec:
      containers:
      - name: fluentd-elasticsearch
        image: jmhobbs/fluentd-elasticsearch:latest
        env:
        - name: CLUSTER_ENDPOINT_URL
          value: https://some-elasticsearch-cluster:9200/
        volumeMounts:
        - mountPath: /var/log
          name: varlog
...
```

A DaemonSet is a pod which should run on every node. A great example of this is a Fluentd log consumer. You want one running on each node so it can tail the logs from Docker and forward them off to storage like Elasticsearch.

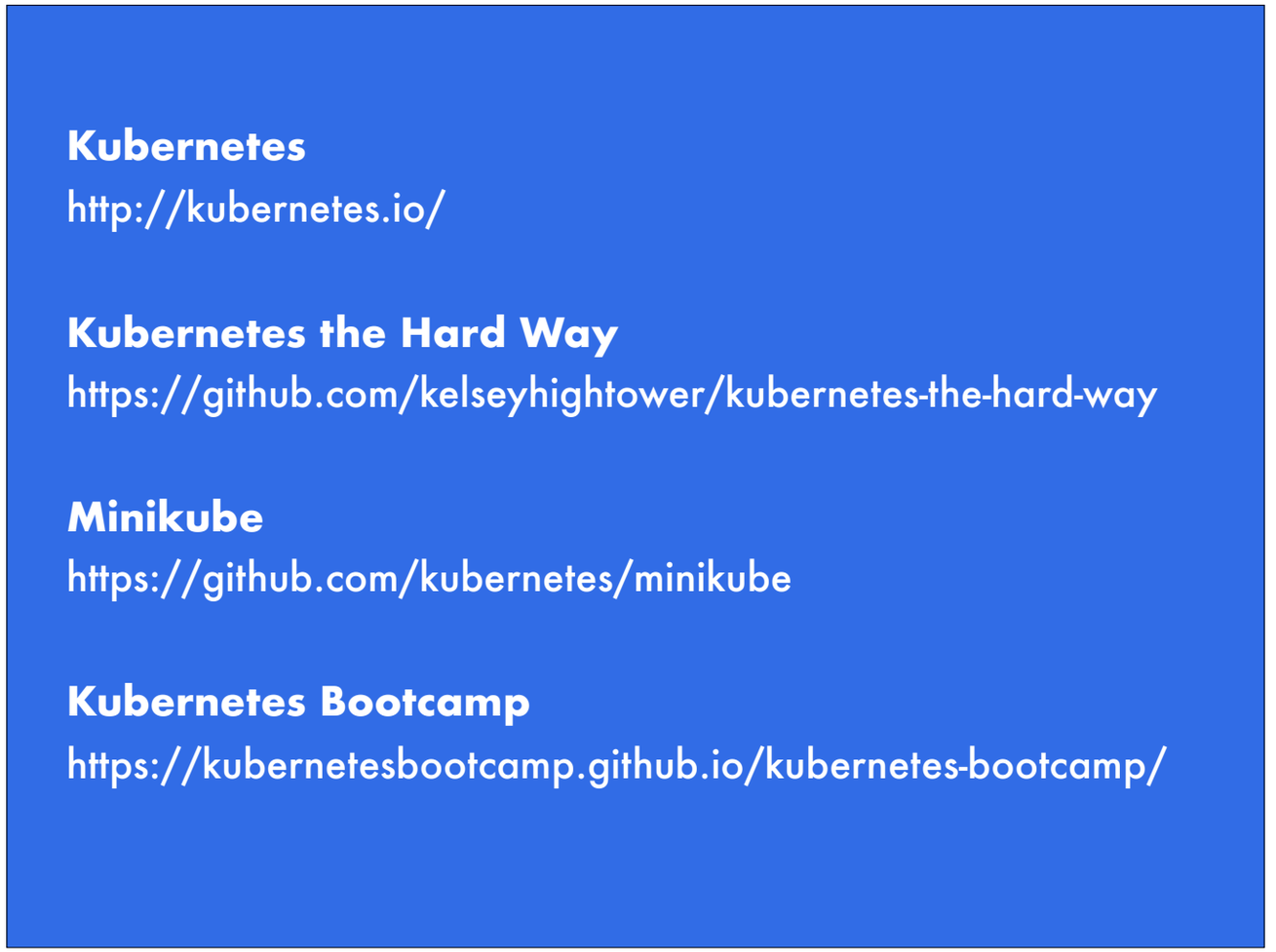
Here is a section of a very large daemonset spec for just that use case.

https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

- Jobs
- Cron Jobs
- Network Policies
- Custom Resource Types
- Federation
- Ingress Controllers
- Scheduling Affinity
- Advanced Authentication Schemes
- Service Accounts

I knew there was no way I could cover everything in this presentation so I left a lot of things out. I hope you will take some time to experiment with it.

https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/
https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/
https://kubernetes.io/docs/concepts/services-networking/network-policies/
https://kubernetes.io/docs/concepts/api-extension/custom-resources/
https://kubernetes.io/docs/concepts/cluster-administration/federation/
https://kubernetes.io/docs/concepts/services-networking/ingress/
https://kubernetes.io/docs/concepts/configuration/assign-pod-node/



Here are some great resources, but the web is just packed with stuff about Kubernetes.

http://kubernetes.io/
https://github.com/kelseyhightower/kubernetes-the-hard-way
https://github.com/kubernetes/minikube
https://kubernetesbootcamp.github.io/kubernetes-bootcamp/

Thanks!

twitter.com/jmhobbs | github.com/jmhobbs | velvetcache.org

https://twitter.com/jmhobbs
https://github.com/jmhobbs
https://www.velvetcache.org
john@velvetcache.org