What is Kubernetes?

- Kubernetes = popular container orchestrator
- Container Orchestration = Make many servers act like one
- Released by Google in 2014, maintained by large community
- Runs on top of Docker (usually) as a set of APIs in containers
- Provides API/CLI to manage containers across servers
- Many clouds provide it for you
- Many vendors make a "distribution" of it

Why Kubernetes?

- Review "Swarm Mode: Built-In Orchestration"
- Orchestration: Next logical step in journey to faster DevOps
- First, understand why you *may* need orchestration
- Not every solution needs orchestration
- Servers + Change Rate = Benefit of orchestration
- Then, decide which orchestrator
- If Kubernetes, decide which distribution
  - cloud or self-managed (Docker Enterprise, Rancher, OpenShift, Canonical, VMWare PKS)
  - Don't usually need pure upstream

Kubernetes or Swarm?

- Review "Swarm Mode: Built-In Orchestration"
- Kubernetes and Swarm are both container orchestrators
- Both are solid platforms with vendor backing
- Swarm: Easier to deploy/manage
- Kubernetes: More features and flexibility
- What's right for you? Understand both and know your requirements

Advantages of Swarm

- Comes with Docker, single vendor container platform
- Easiest orchestrator to deploy/manage yourself
- Follows 80/20 rule, 20% of features for 80% of use cases
- Runs anywhere Docker does:
  - local, cloud, datacenter
  - ARM, Windows, 32-bit
- Secure by default
- Easier to troubleshoot

Advantages of Kubernetes

- Clouds will deploy/manage Kubernetes for you
- Infrastructure vendors are making their own distributions
- Widest adoption and community
- Flexible: Covers widest set of use cases
- "Kubernetes first" vendor support
- "No one ever got fired for buying IBM"
  - Picking solutions isn't 100% rational
  - Trendy, will benefit your career
  - CIO/CTO Checkbox

Basic Terms: System Parts

- Kubernetes: The whole orchestration system
  - K8s "k-eights" or Kube for short
- Kubectl: CLI to configure Kubernetes and manage apps
  - Using "cube control" official pronunciation
- Node: Single server in the Kubernetes cluster
- Kubelet: Kubernetes agent running on nodes
- Control Plane: Set of containers that manage the cluster
  - Includes API server, scheduler, controller manager, etcd, and more
  - Sometimes called the "master"

Install Kubernetes Locally

- Kubernetes is a series of containers, CLI's, and configurations
- Many ways to install, lets focus on easiest for learning
- Docker Desktop: Enable in settings
  - Sets up everything inside Docker's existing Linux VM
- Docker Toolbox on Windows: MiniKube
  - Uses VirtualBox to make Linux VM
- Your Own Linux Host or VM: MicroK8s
  - Installs Kubernetes right on the OS

Kubernetes In A Browser

- Try http://play-with-k8s.com or katacoda.com in browser
  - Easy to get started
  - Doesn't keep your environment

Docker Desktop

- Runs/configures Kubernetes Master containers
- Manages kubectl install and certs
- Easily install, disable, and remove from Docker GUI

MiniKube

- Download Windows Installer from GitHub
- minikube-installer.exe
- minikube start
- Much like the docker-machine experience
- Creates a VirtualBox VM with Kubernetes master setup
- Doesn't install kubectl

MicroK8s

- Installs Kubernetes (without Docker Engine) on localhost (Linux)
- Uses snap (rather then apt or yum) for install
- Control the MicroK8s service via microk8s. commands
- kubectl accessable via microk8s.kubectl
- Add CoreDNS for services to work
  - microk8s.enable dns
- Add an alias to your shell (.bash_profile)
  - alias kubectl=microk8s.kubectl

Kubernetes Container Abstractions

- **Pod**: one or more containers running together on one Node
  - Basic unit of deployment. Containers are always in pods
- **Controller**: For creating/updating pods and other objects
  - Many types of Controllers inc. Deployment, ReplicaSet, StatefulSet, DaemonSet, Job, CronJob, etc.
- **Service**: network endpoint to connect to a pod
- **Namespace**: Filtered group of objects in cluster
- Secrets, ConfigMaps, and more

Kubernetes Run, Create, and Apply

- Kuberentes is evolving, and so is the CLI
- We get three ways to create pods from the kubectl CLI
  - > kubectl run (changing to be only for pod creation)
  - > kubectl create (create some resources via CLI or YAML)
  - > kubectl apply (create/update anything via YAML)
- For now we'll just use run or create CLI
- Later we'll learn YAML and pros/cons of each

Creating Pods with kubectl

- Are we working?

  ```
  > kubectl version
  ```

- Two ways to deploy Pods (containers): Via commands, or via YAML
- Let's run a pod of the nginx web server!

  ```
  > kubectl create deployment my-nginx --image nginx
  ```

- Let's list the pod

  ```
  > kubectl get pods
  ```

- Let's see all objects

  ```
  > kubectl get all
  ```

Pods -> ReplicaSet -> Deployment

Cleanup

- Let's remove the Deployment

```
> kubectl delete deployment my-nginx
```

Scaling ReplicaSets

- Start a new deployment for one replica/pod

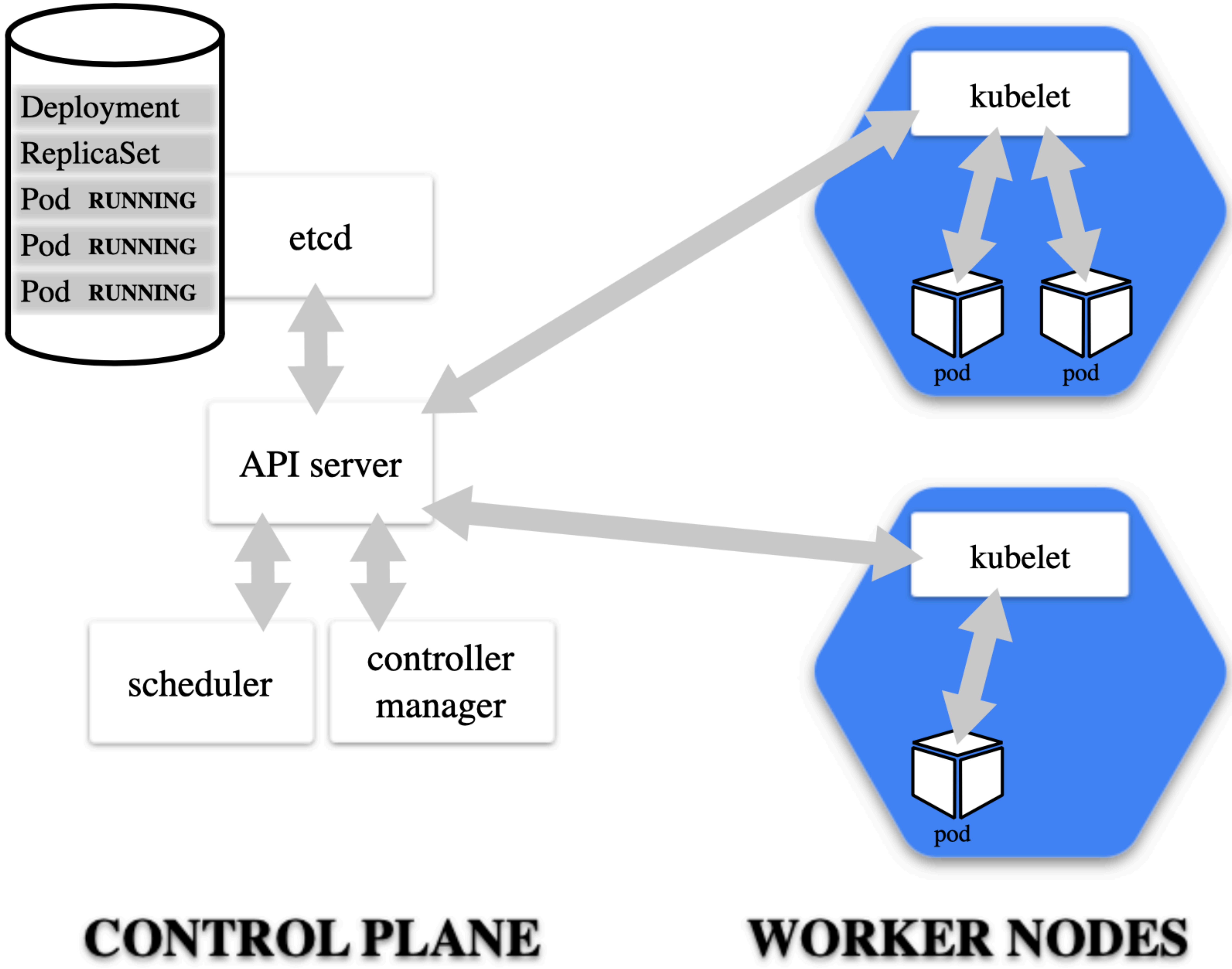
```
> kubectl create deployment my-apache --image httpd
```

- Let's scale it up with another pod

```
> kubectl scale deploy/my-apache --replicas 2
> kubectl scale deployment my-apache --replicas 2
```

  - those are the same command
  - deploy = deployment = deployments

What Just Happened? kubectl scale



Don't Cleanup

- We'll use these httpd containers in the next lecture

Inspecting Deployment Objects

```
> kubectl get pods
```

- Get container logs

```
> kubectl logs deployment/my-apache --follow --tail 1
```

- Get a bunch of details about an object, including events!

```
> kubectl describe pod/my-apache-xxxx-yyyy
```

- Watch a command (without needing watch)

```
> kubectl get pods -w
```

- In a separate tab/window

```
> kubectl delete pod/my-apache-xxxx-yyyy
```

- Watch the pod get re-created

Cleanup

- Let's remove the Deployment

```
> kubectl delete deployment my-apache
```

Exposing Containers

- kubectl expose creates a **service** for existing pods
- A **service** is a stable address for pod(s)
- If we want to connect to pod(s), we need a **service**
- CoreDNS allows us to resolve **services** by name
- There are different types of **services**
  - ClusterIP
  - NodePort
  - LoadBalancer
  - ExternalName

Basic Service Types

- **ClusterIP** (default)
  - Single, internal virtual IP allocated
  - Only reachable from within cluster (nodes and pods)
  - Pods can reach service on apps port number
- **NodePort**
  - High port allocated on each node
  - Port is open on every node’s IP
  - Anyone can connect (if they can reach node)
  - Other pods need to be updated to this port
- **These services are always available in Kubernetes**

More Service Types

- **LoadBalancer**
  - Controls a LB endpoint external to the cluster
  - Only available when infra provider gives you a LB (AWS ELB, etc)
  - Creates NodePort+ClusterIP services, tells LB to send to NodePort
- **ExternalName**
  - Adds CNAME DNS record to CoreDNS only
  - Not used for Pods, but for giving pods a DNS name to use for something outside Kubernetes
- Kubernetes **Ingress**: We’ll learn later

Creating a ClusterIP Service

- Open two shell windows so we can watch this

```
> kubectl get pods -w
```

- In second window, lets start a simple http server using sample code

```
> kubectl create deployment httpenv --image=bretfisher/httpenv
```

- Scale it to 5 replicas

```
> kubectl scale deployment/httpenv --replicas=5
```

- Let's create a ClusterIP service (default)

```
> kubectl expose deployment/httpenv --port 8888
```

Inspecting ClusterIP Service

- Look up what IP was allocated

```
> kubectl get service
```

- Remember this IP is cluster internal only, how do we curl it?
- If you're on Docker Desktop (Host OS is not container OS)

```
> kubectl run --generator=run-pod/v1 tmp-shell --rm -it --image bretfisher/netshoot -- bash
> curl httpenv:8888
```

- If you're on Linux host

```
> curl [ip of service]:8888
```

Cleanup

- Leave the deployment there, we'll use it in the next Lecture

Create a NodePort Service

- Let's expose a NodePort so we can access it via the host IP (including localhost on Windows/Linux/macOS)

  ```
  > kubectl expose deployment/httpenv --port 8888 --name httpenv-np --type NodePort
  ```

- Did you know that a NodePort service also creates a ClusterIP?
- These three service types are additive, each one creates the ones above it:
  - ClusterIP
  - NodePort
  - LoadBalancer

Add a LoadBalancer Service

- If you're on Docker Desktop, it provides a built-in LoadBalancer that publishes the --port on localhost

  ```
  > kubectl expose deployment/httpenv --port 8888 --name httpenv-lb --type LoadBalancer
  > curl localhost:8888
  ```

- If you're on kubeadm, minikube, or microk8s
  - No built-in LB
  - You can still run the command, it'll just stay at "pending" (but its NodePort works)

Cleanup

- Let's remove the Services and Deployment

```
> kubectl delete service/httpenv service/httpenv-np
> kubectl delete service/httpenv-lb deployment/httpenv
```

Kubernetes Services DNS

- Starting with 1.11, internal DNS is provided by CoreDNS
- Like Swarm, this is DNS-Based Service Discovery
- So far we've been using hostnames to access Services

```
> curl <hostname>
```

- But that only works for Services in the same Namespace

```
> kubectl get namespaces
```

- Services also have a FQDN

```
> curl <hostname>.<namespace>.svc.cluster.local
```

Assignment: Explore run get and logs

- Dry Run

```
> kubectl create deployment nginx --image nginx --dry-run
```

- Run does different things based on options

```
> kubectl create deployment nginx --image nginx --dry-run --port 80 --expose
```

- Only create a simple Pod, not a Deployment, ReplicaSet, etc.

```
> kubectl run nginx-pod --generator=run-pod/v1 --image nginx
```

- Get a shell in new Pod, remove on exit

```
> kubectl run shell --generator=run-pod/v1 --rm -it --image busybox
```

Assignment: Explore run get and logs

- Create a Deployment and ClusterIP Dervice in one line

  ```
  > kubectl run nginx2 --image nginx --replicas 2
  ```

- Get multiple resources in one line

  ```
  > kubectl get deploy,pods
  ```

- Get all pods, in wide format (gives more info)

  ```
  > kubectl get pods -o wide
  ```

- Get all pods and show labels

  ```
  > kubectl get pods --show-labels
  ```

  -

Assignment: Explore run get and logs

- Better log viewing with stern
  - github.com/wercker/stern

```
> kubectl run mydate --image bretfisher/date --replicas 3
> kubectl logs deployment/mydate
> stern mydate
```

Cleanup

- Let's remove everything but the service/kubernetes

```
> kubectl get all
> kubectl delete deployment/nginx2 pod/nginx-pod
```

Run, Create, and Expose Generators

- These commands use helper templates called "generators"
- Every resource in Kubernetes has a specification or "spec"

```
> kubectl create deployment sample --image nginx --dry-run -o yaml
```

- You can output those templates with --dry-run -o yaml
- You can use those YAML defaults as a starting point
- Generators are "opinionated defaults"

Generator Examples

- Using dry-run with yaml output we can see the generators

```
> kubectl create deployment test --image nginx --dry-run -o yaml
> kubectl create job test --image nginx --dry-run -o yaml
> kubectl expose deployment/test --port 80 --dry-run -o yaml
```

  - You need the deployment to exist before this works

Cleanup

- Let's remove the Deployment

```
> kubectl delete deployment test
```

The Future of kubectl run

- Right now (1.12-1.15) run is in a state of flux
- The goal is to reduce its features to only create Pods
  - Right now it defaults to creating Deployments (with the warning)
  - It has lots of generators but they are all deprecated
  - The idea is to make it easy like docker run for one-off tasks
- It's not recommended for production
- Use for simple dev/test or troubleshooting pods

Old Run Confusion

- The generators activate different Controllers based on options
- Using dry-run we can see which generators are used

```
> kubectl run test --image nginx --dry-run
> kubectl run test --image nginx --port 80 --expose --dry-run
> kubectl run test --image nginx --restart OnFailure --dry-run
> kubectl run test --image nginx --restart Never --dry-run
> kubectl run test --image nginx --schedule "*/1 * * * *" --dry-run
```

Imperative vs. Declarative

- Imperative: Focus on *how* a program operates
- Declarative: Focus on *what* a program should accomplish
- Example: "I'd like a cup of coffee"
- Imperative: I boil water, scoop out 42 grams of medium-fine grounds, poor over 700 grams of water, etc.
- Declarative: "Barista, I'd like a a cup of coffee". (Barista is the engine that works through the steps, including retrying to make a cup, and is only finished when I have a cup)

Kubernetes Imperative

- Examples: kubectl run, kubectl create deployment, kubectl update
  - We start with a state we know (no deployment exists)
  - We ask kubectl run to create a deployment
- Different commands are required to change that deployment
- Different commands are required per object
- Imperative is easier when you know the state
- Imperative is easier to get started
- Imperative is easier for humans at the CLI
- Imperative is NOT easy to automate

Kubernetes Declarative

- Example: kubectl apply -f my-resources.yaml
  - We don't know the current state
  - We only know what we want the end result to be (yaml contents)
- Same command each time (tiny exception for delete)
- Resources can be all in a file, or many files (apply a whole dir)
- Requires understanding the YAML keys and values
- More work than kubectl run for just starting a pod
- The easiest way to automate
- The eventual path to GitOps happiness

Three Management Approaches

- **Imperative commands:** run, expose, scale, edit, create deployment
  - Best for dev/learning/personal projects
  - Easy to learn, hardest to manage over time
- **Imperative objects:** create -f file.yml, replace -f file.yml, delete…
  - Good for prod of small environments, single file per command
  - Store your changes in git-based yaml files
  - Hard to automate
- **Declarative objects:** apply -f file.yml or dir\, diff
  - Best for prod, easier to automate
  - Harder to understand and predict changes

Three Management Approaches

- **Most Important Rule:**
  - Don't mix the three approaches
- **Bret's recommendations:**
  - Learn the Imperative CLI for easy control of local and test setups
  - Move to apply -f file.yml and apply -f directory\ for prod
  - Store yaml in git, git commit each change before you apply
  - This trains you for later doing GitOps (where git commits are automatically applied to clusters)

kubectl apply

- Remember the three management approaches?
- Let's skip to full Declarative objects

```
>kubectl apply -f filename.yml
```

- Why skip kubectl create, kubectl replace, kubectl edit?
- What I recommend ≠ all that's possible

Using kubectl apply

- create/update resources in a file

  ```
  > kubectl apply -f myfile.yaml
  ```

- create/update a whole directory of yaml

  ```
  > kubectl apply -f myyaml/
  ```

- create/update from a URL

  ```
  > kubectl apply -f https://bret.run/pod.yml
  ```

- Be careful, lets look at it first (browser or curl)

  ```
  > curl -L https://bret.run/pod
  ```

  - Win PoSH? start https://bret.run/pod.yml

Kubernetes Configuration YAML

- Kubernetes configuration file (YAML or JSON)
- Each file contains one or more manifests
- Each manifest describes an API object (deployment, job, secret)
- Each manifest needs four parts (root key:values in the file)

```
apiVersion:
kind:
metadata:
spec:
```

Building Your YAML Files

- **kind:** We can get a list of resources the cluster supports

  > kubectl api-resources

- Notice some resources have multiple API's (old vs. new)
- **apiVersion:** We can get the API versions the cluster supports

  > kubectl api-versions

- **metadata:** only name is required
- **spec:** Where all the action is at!

Building Your YAML spec

- We can get all the keys each **kind** supports

```
> kubectl explain services --recursive
```

- Oh boy! Let's slow down

```
> kubectl explain services.spec
```

- We can walk through the spec this way

```
> kubectl explain services.spec.type
```

- spec: can have sub spec: of other resources

```
> kubectl explain
  deployment.spec.template.spec.volumes.nfs.server
```

- We can also use docs
  - kubernetes.io/docs/reference/#api-reference

Dry Runs With Apply YAML

- New stuff, not out of beta yet (1.15)
- dry-run a create (client side only)

  ```
  > kubectl apply -f app.yml --dry-run
  ```

- dry-run a create/update on server

  ```
  > kubectl apply -f app.yml --server-dry-run
  ```

- see a diff visually

  ```
  > kubectl diff -f app.yml
  ```

Labels and Annotations

- **Labels** goes under metadata: in your YAML
- Simple list of key: value for identifying your resource later by selecting, grouping, or filtering for it
- Common examples include tier: frontend, app: api, env: prod, customer: acme.co
- Not meant to hold complex, large, or non-identifying info, which is what **annotations** are for
- filter a get command

```
> kubectl get pods -l app=nginx
```

- apply only matching labels

```
> kubectl apply -f myfile.yaml -l app=nginx
```

Label Selectors

- The "glue" telling Services and Deployments which pods are theirs
- Many resources use Label Selectors to "link" resource dependencies
- You'll see these match up in the Service and Deployment YAML
- Use Labels and Selectors to control which pods go to which nodes
- Taints and Tolerations also control node placement

Cleanup

- Let's remove anything you created in this section

```
> kubectl get all
> kubectl delete <resource type>/<resource name>
```

Storage in Kubernetes

- Storage and stateful workloads are harder in all systems
- Containers make it both harder and easier than before
- StatefulSets is a new resource type, making Pods more sticky
- Bret's recommendation: avoid stateful workloads for first few deployments until you're good at the basics
  - Use db-as-a-service whenever you can

Volumes in Kubernetes

- Creating and connecting Volumes: 2 types
- **Volumes**
  - Tied to lifecycle of a Pod
  - All containers in a single Pod can share them
- **PersistentVolumes**
  - Created at the cluster level, outlives a Pod
  - Separates storage config from Pod using it
  - Multiple Pods can share them
- CSI plugins are the new way to connect to storage

Ingress

- None of our Service types work at OSI Layer 7 (HTTP)
- How do we route outside connections based on hostname or URL?
- Ingress Controllers (optional) do this with 3rd party proxies
- Nginx is popular, but Traefik, HAProxy, F5, Envoy, Istio, etc.
- Note this is still beta (in 1.15) and becoming popular
- Implementation is specific to Controller chosen

CRD's and The Operator Pattern

- You can add 3rd party Resources and Controllers
- This extends Kubernetes API and CLI
- A pattern is starting to emerge of using these together
- Operator: automate deployment and management of complex apps
- e.g. Databases, monitoring tools, backups, and custom ingresses

Higher Deployment Abstractions

- All our kubectl commands just talk to the Kubernetes API
- Kubernetes has limited built-in templating, versioning, tracking, and management of your apps
- There are now over 60 3rd party tools to do that, but many are defunct
- **Helm** is the most popular
- "**Compose on Kubernetes**" comes with Docker Desktop
- Remember these are optional, and your distro may have a preference
- Most distros support **Helm**

Templating YAML

- Many of the deployment tools have templating options
- You'll need a solution as the number of environments/apps grow
- **Helm** was the first "winner" in this space, but can be complex
- Official **Kustomize** feature works out-of-the-box (as of 1.14)
- docker app and compose-on-kubernetes are Docker's way

Kubernetes Dashboard

- Default GUI for "upstream" Kubernetes
  - github.com/kubernetes/dashboard
- Some distributions have their own GUI (Rancher, Docker Ent, OpenShift)
- Clouds don't have it by default
- Let's you view resources and upload YAML
- Safety first!

Kubectl Namespaces and Context

- Namespaces limit scope, aka "virtual clusters"
- Not related to Docker/Linux namespaces
- Won't need them in small clusters
- There are some built-in, to hide system stuff from kubectl "users"

```
> kubectl get namespaces
> kubectl get all --all-namespaces
```

- Context changes kubectl cluster and namespace
- See ~/.kube/config file

```
>kubectl config get-contexts
>kubectl config set*
```

Future of Kubernetes

- More focus on stability and security
  - 1.14, 1.15, largely dull releases (a good thing!)
  - Recent security audit has created backlog
- Clearing away deprecated features like kubectl run generators
- Improving features like server-side dry-run
- More and improved Operators
- Helm 3.0 (easier deployment, chart repos, libs)
- More declarative-style features
- Better Windows Server support
- More edge cases, kubeadm HA clusters

Related Projects

- Kubernetes has become the "differencing and scheduling engine backbone" for so many new projects
- Knative - Serverless workloads on Kubernetes
- k3s - mini, simple Kubernetes
- k3OS - Minimal Linux OS for k3s
- Service Mesh - New layer in distributed app traffic for better control, security, and monitoring