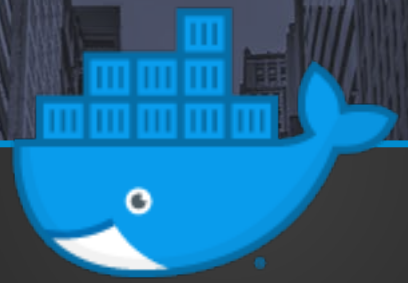
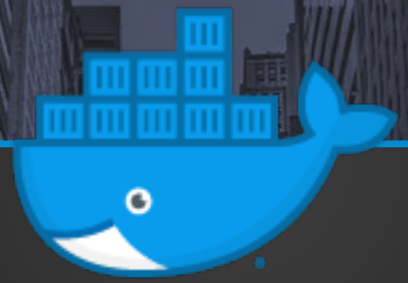
KUBERNETES FUNDAMENTALS

KUBERNETES INSTALL



INTRO TO ORCHESTRATION AND KUBERNETES

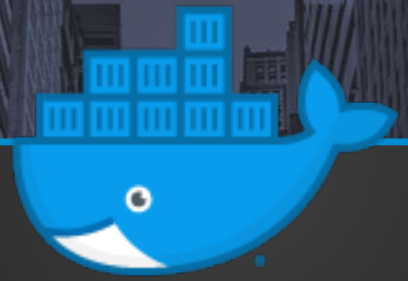
ORCHESTRATION



ORCHESTRATION - OVERVIEW

- What's wrong with just running containers using Docker run in production? Remember: we're doing microservices now, with lots of services! And we need redundancy. So at a minimum we're talking about dozens of running containers for even a basic application!
- Why not just run Compose on servers?
- Healing and scaling
- How do you manage hundreds of servers with thousands of containers?
- What options are out there? Swarm, ECS, Kubernetes, Mesosphere.
- Discuss clusters, health monitoring, services, service discovery, geographic distribution, load balancing.
- What is a control plane or scheduler?
- Managed deployments

KUBERNETES BASICS



KUBERNETES — A BRIEF HISTORY

- 2003 – Started at Google as The Borg System to manage Google Search. They needed a sane way to manage their large-scale container clusters! But it was still very primitive compared to what we have today. It becomes big and rather messy, with many different languages and concepts due to it's organic growth.
- 2013 –Docker hits the scene and really revolutionizes computing by providing build tools, image distribution, and runtimes. This makes containers user friendly and adoption of containers explodes.
- 2014 – 3 Google engineers decide to build a next generation orchestrator that takes many lessons learned into account, built for public clouds, and open sourced. They build Kubernetes. Microsoft, Red Hat, IBM, and Docker join in.
- 2015 – Cloud Native Computing Foundation is created by Google and the Linux Foundation. More companies join in. Kubernetes 1.0 is released, followed by more major upgrades that year. KubeCon is launched.
- 2016 – K8S goes mainstream. Many supporting products are introduced including Minikube, kops, Helm. Rapid releases of big features. More companies join in and the community of passionate people explodes.
- 2017 to now – Kubernetes becomes the dominant orchestration system and de-facto standard for Docker microservices. K8S now has fully managed services by all major cloud providers. Handsome and talented instructors travel the country preaching K8S. ☺

KUBERNETES — THE ALTERNATIVE

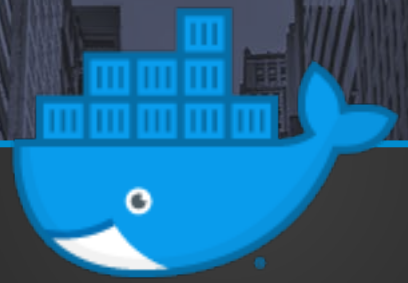
Before cloud and Docker orchestration came along, production support and releases was a wild west full of danger and long nights.

Releases were a MAJOR nightmare for many. Spending most/all of a night on the phone with an entire ops team supporting a release and fixing issues was commonplace. The build pipeline was slow and prone to errors, and frequently took days to complete. When releases failed, you “failed forward” instead of rolling back.

Let me tell you about my own experiences before containerized workloads.

Let’s also talk through some stories the class has of painful releases. We’ll try to talk about how those issues are resolved using orchestration.

KUBERNETES ARCHITECTURE



KUBERNETES – DECLARATIVE MODEL

K8S utilizes infrastructure-as-code concepts. All resources and configurations are done in declarative YAML files called resource templates or manifests and applied to your cluster.

Example resource:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
  spec:
    containers:
    - name: myapp-container
      image: busybox
      command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

KUBERNETES – MINIKUBE

Minikube is a small K8S cluster intended to run on your local machine. We will spend lots of time here.

It provides much of the functionality that a full cluster has. You can run a completely functional application on your machine.

Minikube commands:

- Minikube start
- Minikube stop
- Minikube ip – retrieve the IP of your cluster for use in a browser
- Minikube dashboard – bring up the web UI dashboard

KUBERNETES — DOCKER KUBERNETES

New in Docker, you can now enable Kubernetes on your workstation with a simple checkbox. You'll find it in Preferences.

Unlike Minikube where a virtual machine is launched, you can access the services using localhost instead of the VM IP address.

You can disable and shut down your local cluster using the Docker menu.

KUBERNETES – KUBECTL

Kubectl provides the command line interaction with your cluster. This is how you'll view and manage things like pods, services, etc, and apply template files to your cluster. Commands are formatted as `kubectl <command>`

Commands:

- Get <service, pod, etc> <name> - get information about the resource
  - `kubectl get pod mypod`
- Get all – return a full page of details about your cluster's resources
- Apply – apply a template file to your cluster
  - `kubectl apply -f workloads.yaml`
- Describe – get detailed information about a resource
  - `kubectl describe service myservice`
- Logs – return console output from a pod (or a container inside it)
  - `kubectl logs mypod`

KUBECTL – CONTEXTS

Kubectl can be pointed to different clusters for management. If you installed minikube and also activated Kubernetes for Docker, you'll need to tell kubectl which cluster to work on. You do this with context.

- kubectl config view – shows config information, including available contexts
- kubectl config current-context – shows the current context
- kubectl config use-context docker-for-desktop – switch kubectl to send commands to the cluster identified by the name "docker-for-desktop"

You can also switch contexts using the Docker menu by expanding the Kubernetes item.

KUBERNETES — CORE COMPONENTS

- Master components – The controller for the cluster. Also known as the "control plane"
- Nodes – the servers in our cluster. There are master nodes and worker nodes.
- Workloads – running containers to do the work. These can be created in several ways as we'll see.
- Services – provide network connectivity to workloads.

KUBERNETES – MASTER COMPONENTS

- Master components are pods themselves that run in the kube-system namespace
- Master node – a special node to hold many of the master components. For fault tolerance, 3 of these should be used for production.
- Kube-apiserver – exposes the K8S API
- etcd – key-value store to hold cluster data
- kube-scheduler – places pods onto nodes

Note that the loss of the master node does not mean that your cluster is down! No management will happen until the master is back up, but it is not a gateway for traffic to your services.

KUBERNETES – WORKER NODES

- Worker servers that run your workloads.
- Several overhead pods run on every node (these are master components):
  - Kubelet – an agent pod that ensures containers are running in a pod
  - Kube-proxy – a pod that maintains network connections/routing
  - Other optional pods and controllers (ie. Logging)
- Has the container runtime (ie. Docker)

PODS

KUBERNETES — PODS

- The basic building block of K8S. It represents a running process in your cluster.
- It encapsulates a container. In rare cases, it can contain multiple containers that are tightly coupled and must run together.
- Provides additional configuration about how the container runs.
- K8S manages pods, not containers.
- Every pod gets a unique IP.
- If multiple containers per pod, pods can communicate with localhost.
- You will rarely interact directly with pods, except perhaps viewing logs. You will interact with Deployments or ReplicaSets instead.
- When a pod dies, no replacement is automatically created, unless it was created as part of a ReplicaSet or Deployment.
- Pods are private by default (from outside the cluster). How do you get your web server accessible? That's coming up next with a service.
- Pods get a label, which we'll learn more about later.
- Let's launch a pod and examine it.

KUBERNETES — POD SIDECAR

- Usually you want to have a pod contain just a single container.
- However, there are cases when you'd want to have multiple containers in a single pod.
- The sidecar pattern is an example of this.
- With a sidecar, you run a second container in a pod whose job is to take action and support the primary container.
- Logging is a good example, where a sidecar container sends logs from the primary container to a centralized logging system.

KUBERNETES – POD EXERCISE

1. Exercise file: exercises/Day 4/pod.yaml
2. Let's apply it
3. Review all pods
4. Describe the pod
5. How do we access port 80? We can't.
6. Let's launch a bash shell inside the pod and test it.
    1. kubectl exec -it nginx-pod -- /bin/bash
    2. curl http://localhost. Oh no! curl not found!
    3. Apt-get update
    4. Apt-get install curl
    5. Now try again
    6. We should now get the HTML for the nginx default page
7. Exit and leave the pod running
8. Side note: if I launch this pod again, will curl be there?

SERVICES

KUBERNETES – SERVICES

- Pods are mortal and unstable. Especially with ReplicaSets, pods can frequently be shuffled out of service, especially with scaling events. Services step in to provide a single, stable point of entry to your pods.
- Pods attached to a service are specified using label selectors
- Services are how we provide traffic access to pods.
- Services act as load balancers for our pods. They just distribute traffic in a round-robin pattern.
- Service types:
  - ClusterIP – default option. Provides access inside the cluster. Good for services that should remain internal to the application like a database.
  - NodePort – Provides a port outside of the cluster. Makes your service public. Note: the port you expose on the cluster is only valid from 30000-32767
  - LoadBalancer – Only valid on cloud implementations, and creates a load balancer and attaches it to your service.
- Let's give this a go with an exercise.

KUBERNETES — LABELS AND SELECTORS

- Kubernetes uses selectors to identify pods that will be included in a service
- These are name/value pairs that you define.
- The scheme you use is up to you.
- You can have multiple selectors and all must match.
- You can define elaborate schemes to enable sophisticated deployment mechanisms such as blue/green.

KUBERNETES – SERVICES EXERCISE

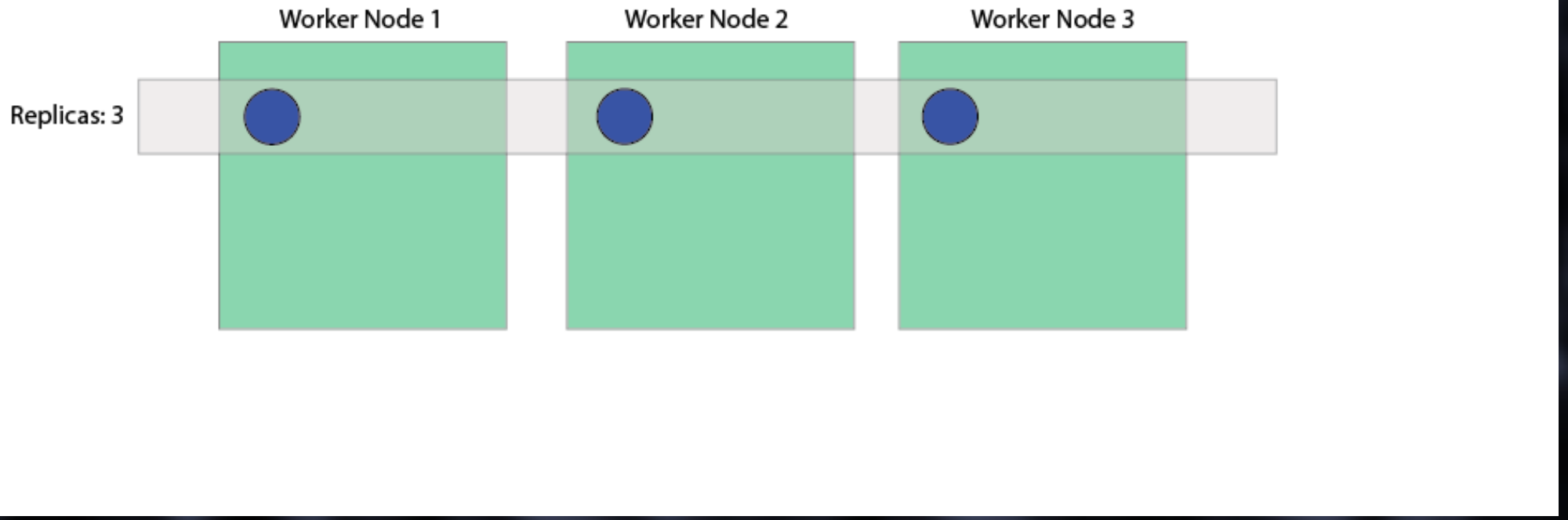
1. Exercise file: exercises/Day 4/service.yaml
2. Review the file. Pay attention to the type and ports.
3. Apply to cluster
4. Now we can access the service from our own computer at http://localhost:30080
5. Now go delete your pod and try the URL again. Oh no! Our service is down!
6. Don't delete the service. We'll use it again shortly.

REPLICASETS

KUBERNETES – REPLICASET

- When we deleted the pod in the last exercise, our service went down!
- We don't want to manage pods directly, because they're mortal
- Instead we work with replicasets to manage the creation and replacement of pods.
- With a replicaset, any pods that die will be recreated to maintain the minimum number.
- This is essentially our pod declaration with some additional meta data.
- Let's go create one.

KUBERNETES — REPLICASET



KUBERNETES — REPLICASET

So what component in Kubernetes makes sure the specified number of replica pods are running? The answer shows the elegance of Kubernetes' architecture.

1. The ReplicationController (which is the logic behind the ReplicaSet resource) is always watching the K8s API. When it sees that the number of running pods differs from the ReplicaSet's configuration, it takes action. It tells K8s to launch pods to make up the gap. The ReplicationController's job is now done.
2. The pod records are created in etcd.
3. The Scheduler now starts finding nodes for the pods to run.
4. When a pod is assigned to a node, the kubelet on the assigned node takes action to start the pod on the node.

KUBERNETES – REPLICASET LAB

1. Exercise file: exercises/Day 4/replicaset.yaml
2. Review the file. Note the labels and replicas. Does the spec section look familiar?
3. Apply the file to your cluster.
4. Kubectl get all. Note that we now have pods, services, and replicasets
5. We now have our nginx pod running again! Try the URL to the service.
6. Now go delete your pod. Does the service go down? Does it go down temporarily? Do a kubectl get all and note the name of the new pod.
7. Now go increase the replicas setting in the template file and reapply.
8. Repeat the pod delete step, and note that you now have a resilient service that auto heals but also has multiple pods to serve requests!
9. When done, delete the replicaset.
10. How do we handle updates to your container images, such as new version releases? Let's talk deployments.

DEPLOYMENTS

DEPLOYMENTS – OVERVIEW

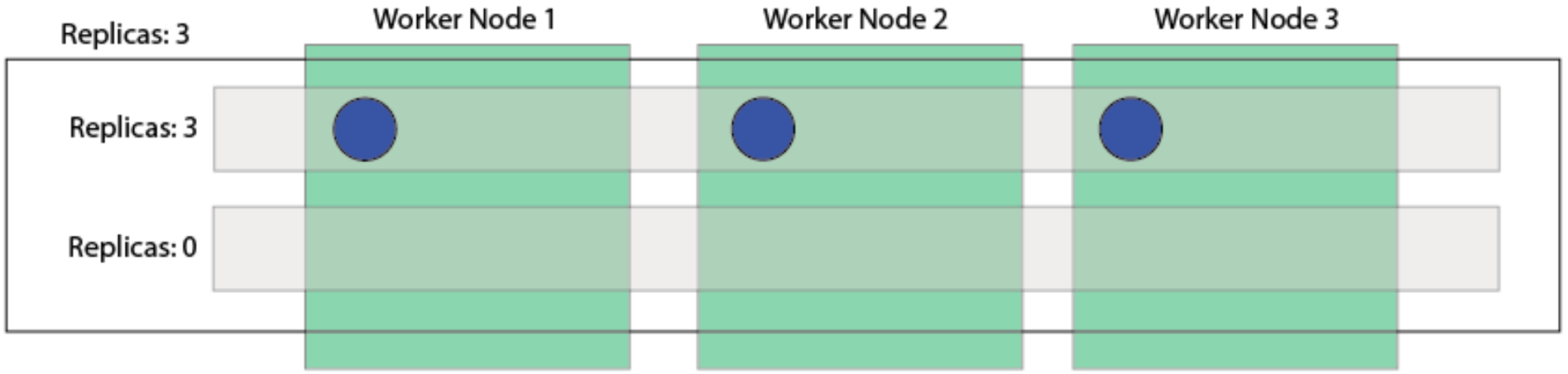
- Deployments are an abstraction that creates ReplicaSets and manages pods being launched into different ones in a controlled way.
- They are declarative and you provide the end state. Kubernetes takes care of getting your pods to that end state in a managed way.
- On a rollout, K8S creates a new replicaset and starts moving pods to the new one in a controlled way, before removing the old replicaset.
- Note: deployments create a replicaset programmatically. Do not manage the replicaset directly!
- You can see rollout status with: kubectl rollout status deployment nginx-deployment
- You can see rollout history with: kubectl rollout history deployment nginx-deployment
- You can rollback with: kubectl rollout undo deployment nginx-deployment
- You can pause and resume rollouts
- There are tons of settings here: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

DEPLOYMENTS — ROLLOUTS

When you update the pod definition in your Deployment, the DeploymentController will start a rolling update process. It does this by simply managing ReplicaSets for you. Assume you already have code running in your Deployment.

1. A new ReplicaSet is created with the new Pod configuration. The Replicas count is zero.
2. The Replicas count will be increased on the new ReplicaSet.
3. Once the pods are launched, the Replicas count on the original ReplicaSet are reduced.
4. This process will continue until the new ReplicaSet has the original Replicas count and the old ReplicaSet has a Replicas count of zero.
5. The old ReplicaSet will hang around empty. By default 10 (which is customizable in your Deployment spec). A rollback will reverse this process.

DEPLOYMENTS — OVERVIEW



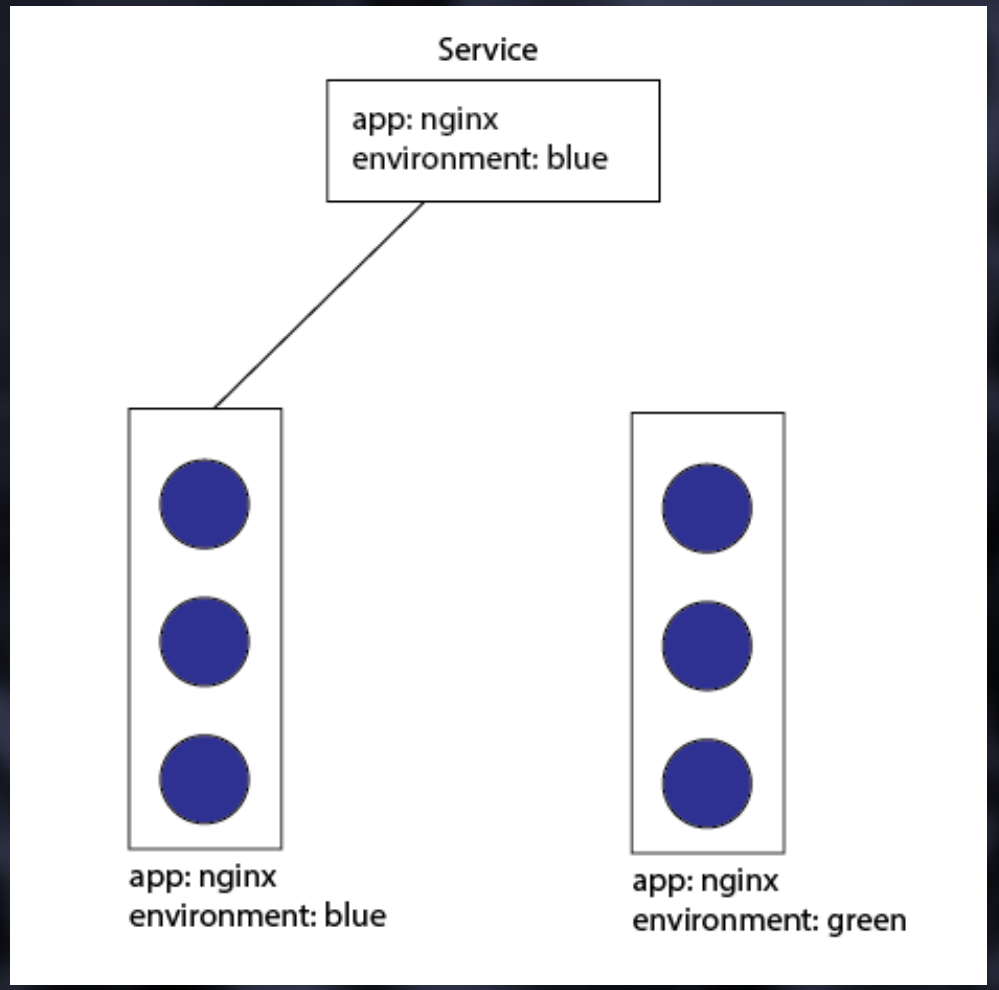
DEPLOYMENTS – EXERCISE

1. Exercise file: exercises/Day 4/deployment.yaml
2. Review the file. Does it look just like a replicaset? Sure does! Note the image tag of 1.0.
3. Apply it to the cluster. Kubectl get all and note we now have pods, services, replicasets, and deployments. Your service should be back up at your cluster IP URL and will behave the same way as your replicaset did.
4. Now update the deployment file to upgrade nginx from 1.0 to 2.0. When you apply it, quickly kubectl get all, and notice how a new replicaset and pods are created. At the URL, if you keep refreshing, you'll see version 1 change to version 2 without the service ever going down.

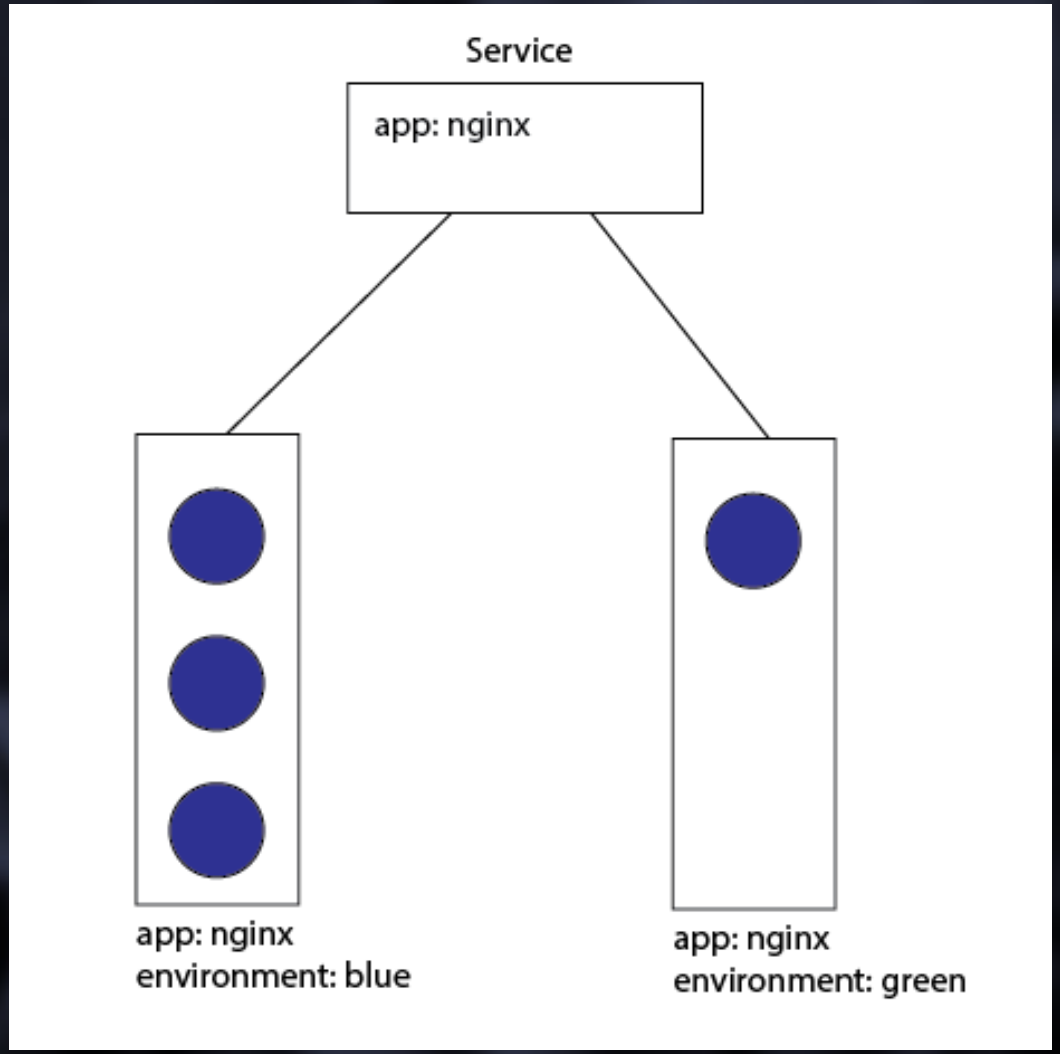
DEPLOYMENTS — DEPLOYMENT METHODS

- Kubernetes supports rolling deployments, which is default. But sometimes applications can't work with that type of deployment and need a full cutover. You need another option.
- Blue/green is a common pattern, where an entire new copy of the application is created, traffic is drained to the old deployment, and then switched over all at once.
- Blue/green is not supported by K8S. But you can still implement it yourself manually or with some scripting.
- You can also do canary deployments in the same way as blue/green, but by creating a small Deployment of the new code, using common labels in your Service so that the canary pods are mixed in with original pods. Once the code is vetted, then the canary Deployment can be expanded and the original Deployment reduced.

BLUE/GREEN DEPLOY — OVERVIEW



CANARY DEPLOY — OVERVIEW



BLUE/GREEN DEPLOY – LAB

- You can perform a blue/green deployment by using a label scheme with more than one set of labels.
- In the exercise files, the day 4 directory, you'll find two files you can use: service.yaml and deployment.yaml.
- Modify the deployment to have a label for version. Then modify the service to match.
- Apply the service and deployment. Now create a copy of the deployment, but use version 2.0 of the Docker image.
- Now apply this deployment and then do a full blue-->green cutover at once without doing a rolling deployment.
- When done, I'll demo the solution to make sure everyone understands.

DEPLOYMENTS – ADVANCED CONFIGURATION

- We can tweak the settings of Deployments to customize their behavior.
- In the `spec`, you can specify a `strategy` for replacing old pods with new ones. The options are `Recreate` or `RollingUpdate` (default).
- With `Recreate`, all old pods are killed off before the new ones are launched. This means you'll experience downtime.
- For `RollingUpdate`, you can adjust the behavior.
  - `maxUnavailable` specifies how many pods can be unavailable at any time during rollout. You can specify absolute number or percentage. The default is 25%.
  - `maxSurge` is how many pods can be created over the replicas count. Can be absolute or percentage. The default is 25%.
- `minReadySeconds` specifies a waiting period before a new Pod is considered ready. For a better solution, use Probes.

VOLUMES

KUBERNETES – VOLUME

- The filesystem in a container is ephemeral, because containers are immutable. Volumes provide persistent storage on the host system.
- Volumes can be a variety of types, from the host hard drive to cloud storage volumes like AWS EBS.
- To read up on the details: https://kubernetes.io/docs/concepts/storage/volumes/
- K8S volumes include lots of management automatically, such as mounting your EBS volumes to pods and unmounting them.
- This topic can get messy. We're going to dive into more depth on day 5 when we create our infrastructure on AWS.

KUBERNETES – VOLUME

- There are a few key resource types:
  - PersistentVolume – defines a storage volume in your cluster. They live separately from any workloads and have a management lifecycle completely separate.
  - StorageClass – defines what “classes” of storage you’ll offer in your cluster. When used, provides a dynamically created PersistentVolume.
  - PersistentVolumeClaim – are a request for storage from a PersistentVolume. Can be part or all of the PersistentVolume. You’re making a “claim” on a volume.

KUBERNETES – NAMESPACE

- These are a way to create virtual clusters on the same physical clusters.
- For example, you could create dev and test environments on the same K8S cluster hardware.
- You likely won't need to worry about this much.
- You should know about the kube-system namespace. System level stuff goes here.
- The default namespace where all your stuff goes if you don't tell K8S otherwise.
- Everything we've done so far was in the default. Later we'll add a few resources to the kube-system namespace.

PROBES

PROBES – OVERVIEW

- Pods and their containers can misbehave. We need to have a way to control how Kubernetes handles them.
- When a pod becomes unhealthy, we want Kubernetes to be able to restart it.
- Because pods and their containers don't start up immediately, we want Kubernetes to hold traffic from hitting the pod until it is ready.
- There are two kinds of probes in Kubernetes to solve these problems: liveness and readiness.
- You can combine liveness and readiness probes for a robust pod.
- You have some options to fine tune (they all have defaults):
  - initialDelaySeconds
  - periodSeconds – default 10
  - timeoutSeconds – default 1
  - successThreshold – default 1
  - failureThreshold – default 3

PROBES – LIVENESS PROBES

- Kubelet uses liveness probes to know when to restart a container.
- Once Kubernetes senses that a pod is unhealthy, it will restart it.
- Liveness probes can be different types: TCP, HTTP, filesystem check

PROBES – READINESS PROBES

- Kubelet uses readiness probes to know when to start sending traffic to a container. A pod is ready when all containers are ready. You can use this to prevent pods from accepting traffic before they're fully initialized.

PROBES – LAB 1

In this exercise, a nginx container starts, but after 30 seconds the index.html file is removed, which causes nginx to no longer return a 200 status code. The probe will pick up on that and restart the container, starting the process over.

1. Exercise file: exercises/Day 4/probe.yaml
2. Review the template file. Notice there are two resources in this one!
3. Apply to your cluster.
4. Open a browser and verify that you can access it.
5. Kubectl get all. Notice the pod's restarts column.
6. Kubectl describe pod liveness-http, and notice the output shows the restarts
7. When done, delete the resources using the template file. Kubectl delete –f probe.yaml

PROBES — LAB 2

Now let's add a readiness probe so that traffic doesn't get sent to a down pod. This lab includes the same pod from the last lab, but as a deployment with several copies. It has an initContainer (coming up) that randomly sleeps before starting the main pod.

1. Exercise file: exercises/Day 4/probe-2.yaml
2. Review the template file.
3. Apply to your cluster.
4. Open a browser and verify that you can access it.
5. After about a minute, you should no longer get a valid response from any pod, because they all get the index.html file removed.
6. Now update the deployment to contain both a readiness probe (to remove the pod from service when it goes down) and a liveness probe (to restart the container when it fails).
7. Run a watch kubectl get all, and watch the pods as they restart. Notice how the status goes back and forth between `1/1` and `0/1`.
8. In a new terminal tab, run a watch kubectl describe service nginx-service. Notice the endpoints shifting as the service shuffles pods in and out of service based on their readiness probe status.
9. You should now have a resilient service that stays up. It might occasionally go down if all three pods are broken at the same time. But you get the idea.

KUBERNETES – CONFIGMAP

- Containers often need several environment variables to function properly so that configuration is externalized. But passing in variables directly to containers is messy.
- ConfigMaps allow you to decouple configurations at the cluster level and have containers refer to them.
- This also allows you to reuse configurations.

KUBERNETES – CONFIGMAP EXERCISE

1. Exercise file: exercises/Day 4/configmap.yaml
2. Review the file and apply to your cluster
3. Take a look at the log output for the MongoDB pod
4. Also describe the MongoDB pod and notice the output references the configmap
5. When done, delete the resources

KUBERNETES — SECRET

- How do you store sensitive information? Should you include it in a Docker image? How about in a pod spec? Never!
- Secrets are small pieces of sensitive information that your pods can access at runtime. Think passwords, SSH keys, etc.
- Secrets are stored as volumes that your containers can access. Alternatively, they can be exposed as environment variables.
- When using kubectl get, you won't see the contents of a secret.
- Note that secrets are still accessible to those with access directly to the cluster. They are meant to protect from including them in Docker images which are more portable. It is best to have secrets managed by a limited set of people who know how to keep them safe. And don't just check them into source control alongside your resources.
- When secrets are updated, the containers automatically pick up the changes immediately.

KUBERNETES – SECRET EXERCISE

1. Exercise file: exercises/Day 4/secret.yaml
2. Review the file and apply to your cluster
3. Take a look at the log output for the MongoDB pod
4. Also describe the MongoDB pod and notice the output references the secret. Also notice that it was mounted as a volume to the container for you.
5. When done, delete the resources

Note: for simplicity, the secret and deployment are together. Don't do this in a real world scenario.

KUBERNETES – DAEMONSET

- Runs a copy of the pod on every node in the cluster
- Any new node will get a new copy of the pod
- Any node removal cleans up the copy of the pod
- Useful for system level resources such as monitoring, logging, etc.
- This is how the master pods on the worker nodes run, such as the kube-proxy and kubelet.

KUBERNETES — STATEFULSET

- Works like a deployment, but provides guarantees about the order and uniqueness of pods
- Pods get a consistent naming scheme that is ordered. For example, pod-0, pod-1, pod-2, etc.
- The spec is identical, except for the Kind statement
- Stable, persistent storage
- Not typical. You should aim for stateless components if possible and use Deployments instead.
- Useful when you have a group of servers that work together and need to know each others' names ahead of time. For example, we will create an ElasticSearch cluster later that uses StatefulSets.

KUBERNETES – JOB

- Jobs are short-lived processes that create pods to fulfill the work and then cleanup the pods when done.
- You can create jobs programmatically, or have timed jobs with CronJobs.
- Great for batch operations. Think daily import processes or perhaps an inventory management process that runs periodically. Maybe a backup job.
- Keep in mind that in certain situations the schedule can create multiple jobs based on one CronJob, so design accordingly.

KUBERNETES – JOB EXERCISE

1. Exercise file: exercises/Day 4/job.yaml
2. Review the file and apply to your cluster
3. Review the pods in the cluster, look at output, etc.
4. Delete when done

KUBERNETES – OBSERVING RESOURCES

- You can observe your Kubernetes resources in a variety of ways.
- Kubectl is the centerpiece and allows you to output resources in different formats using the –o flag.
- To see the help for the output flag, try kubectl get --help
- Try viewing some resources in yaml, json, wide, etc.
- You can also view resources according to their label with the –l flag in your kubectl get commands.
- To get the resource template for a resource, do a get with the --export flag. This will give you yaml that you can edit and reapply.
    - For example kubectl get pod mypod –o wide --export

TROUBLESHOOTING IN

KUBERNETES

KUBERNETES – DEBUGGING

- A huge part of learning Kubernetes is learning how to systematically troubleshoot issues.
- You must understand how all of the pieces work together to work through any issues.
- You need to be adept at observing the current state of the cluster and applications to determine what needs to be fixed.

KUBERNETES – TROUBLESHOOTING LAB

In this lab, a service is down. The team has asked for your help to debug and correct the issue.

1. Without looking at the file, apply the file exercises/day 4/troubleshooting.yaml to your cluster.
2. Find the broken pod. Determine what is causing the issue.
3. Export the yaml definition for the pod, make the necessary changes, and apply it to the cluster to fix the broken pod.
4. Try to bring up the service in your browser. To find the port, examine the service.
5. Figure out why the service is broken.
6. Export the resource template and fix the problem so that the application runs successfully in the browser.

SECURITY

BASICS

NETWORK

POLICIES

KUBERNETES – NETWORK POLICIES

- Pods are wide open by default. If the IP address is known, any other pod within your cluster can access them.
- Network policies allow you to control traffic flow to your pods.
- Like services, network policies select their pods using label/selectors.
- Inbound (ingress), outbound (egress), and both directions can be controlled.
- They operate as a whitelist. Once a policy is applied to a pod, only the traffic specified will be allowed. However, it depends on ingress/egress.
  - For example, if a type of ingress is supplied, then only the inbound traffic specified will be allowed. Because egress wasn't included, all outbound traffic is allowed.
- Policies can also control traffic from certain namespaces or IP address blocks (using CIDR blocks).
- Multiple policies can be used. They do not conflict or cancel each other out. Their rules simply add together to form a complete policy.
- Not all networking plugins support network policies.

KUBERNETES – NETWORK POLICY LAB

- Let's test out a network policy using Nginx and a client pod.

SECURITY CONTEXTS