A Technical Report on Kubernetes

SoSe 2023

Contents

1 Introduction to Kubernetes **2**
1.1 The History of Kubernetes 2
1.2 The Role of Kubernetes 2

2 Kubernetes Architecture **2**
2.1 Overview 2
2.2 Master 3
2.3 Nodes 3
2.4 Pods 4
2.5 API and API Server 4

3 Setting up Kubernetes Cluster **5**
3.1 Minikube 5
3.2 Kubeadm 5
3.3 Cloud Provider Managed Kubernetes 6

4 Kubernetes Pods **6**
4.1 Deployment Workflow 6
4.2 Deploying a Pod 6

5 Kubernetes Services **8**
5.1 NodePort Service 8
5.1.1 Creating NodePort Imperatively 8
5.1.2 Creating NodePort Declaratively 9
5.2 LoadBalancer Service 10

6 Kubernetes Deployments **10**
6.1 Replica Sets 10
6.2 Update and Rollback 11
6.3 Deployment YAML File 12

7 Comparison to Docker Swarm **13**
7.1 Difference between Docker Swarm and K8s 13
7.2 Examples from Docker Swarm lecture in K8s 13

1 Introduction to Kubernetes

1.1 The History of Kubernetes

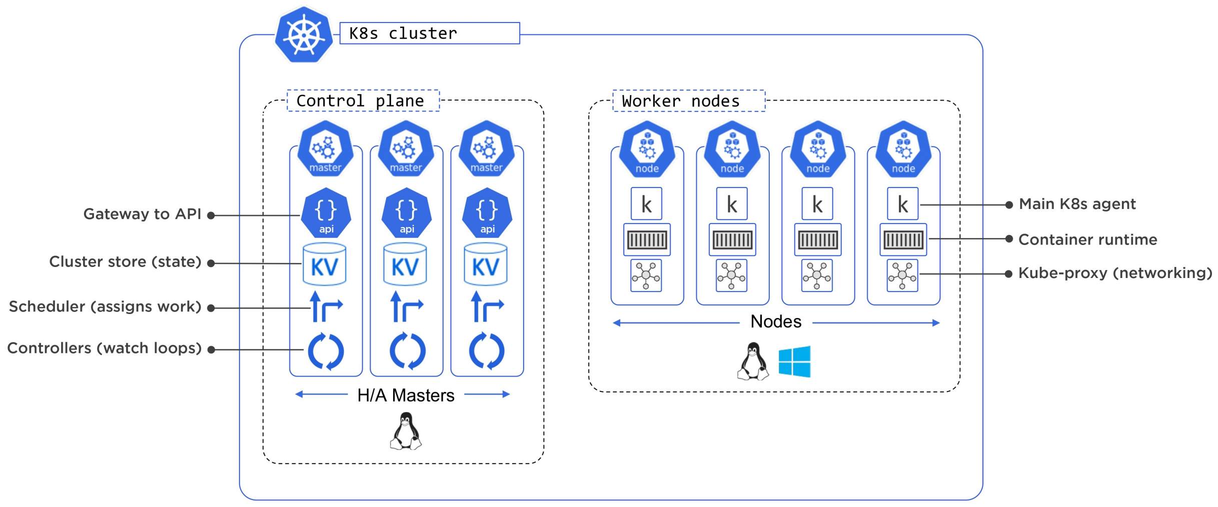
Kubernetes originated from Google and was initially released as an open-source project in the summer of 2014. Subsequently, it was entrusted to the Cloud Native Computing Foundation. Since then, Kubernetes has experienced an extraordinary rise in popularity, emerging as one of the most prominent and influential open-source technologies worldwide. Typical of innovative cloud infrastructure initiatives, Kubernetes is implemented using the Go programming language, commonly referred to as Golang. It is worth noting that Kubernetes is often abbreviated as "K8s," where the number 8 is employed as a substitution for the eight characters occurring between the letters "K" and "s".

1.2 The Role of Kubernetes

In the context of application modernization and management, it is evident that the transition from legacy virtual machine-based applications to containerized applications necessitates a significant increase in the number of containers. Consequently, effective management of these containerized applications becomes imperative. In light of this, Kubernetes emerges as a valuable solution.
The fundamental premise is to interact with Kubernetes, informing it about the application and its constituent services, and requesting it to handle the execution and orchestration on our behalf. Kubernetes fulfills this role, relieving us of the burden of intricate decision-making and implementation details.

2 Kubernetes Architecture

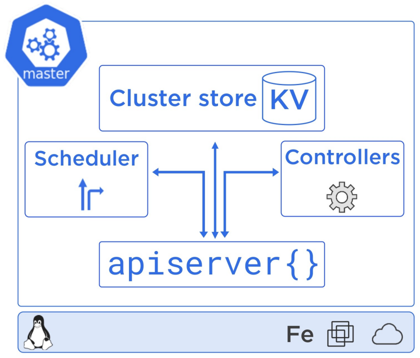
2.1 Overview



The initial architecture of our application consists of various individual services, each encapsulated within a distinct container. These services range from load balancers and web servers to logging systems, forming a comprehensive and diverse ecosystem. The orchestration of these services is facilitated by Kubernetes.
The application is deployed in a Kubernetes-managed cluster, with *master nodes* controlling scheduling and *worker nodes* running applications. For deployment, the application code is containerized and encapsulated within a *pod* and further wrapped inside a *deployment* for enhanced functionalities. The application's specifications are detailed in a *Kubernetes YAML file*, which is given to the master node, showcasing the power of Kubernetes.

2.2 Master

The masters, often referred to as the control plane or head nodes, represent the intelligence of the Kubernetes cluster. Since they govern the operation of the cluster, ensuring their constant availability is crucial. To maintain high availability, an odd number of masters are utilized, and they are distributed across different failure domains linked by robust, reliable networks. In most cases, three masters are deemed sufficient. Deploying two masters is discouraged due to the potential risk of a split-brain condition and deadlock.
The master nodes, which must be Linux machines, can be located virtually anywhere and run multiple smaller services, each responsible for a single control plane feature. These master components can be managed by a user in a self-built cluster or hidden and managed by a cloud provider in a hosted Kubernetes platform. However, user applications should typically run on worker nodes, not on the masters, to ensure clean demarcation and simplicity.



The master's key components include the *API server*, the *cluster store*, the *controller manager*, and the *scheduler*. The API server is the gateway to the cluster and the medium through which commands are sent to the cluster. The cluster store is the sole persistent component, storing the state and configuration of the cluster and apps. The controller manager oversees a variety of controllers responsible for different cluster elements, and the scheduler assigns workloads to nodes while considering several factors for optimal distribution.

2.3 Nodes

This section covers the fundamental aspects of Kubernetes nodes. The core components of interest are the **kubelet**, the **container runtime**, and the **kube-proxy**.
The kubelet is the primary Kubernetes agent running on each cluster node. Nodes, often referred to interchangeably with kubelets, begin as Linux or Windows machines. Once the kubelet is installed on a machine, it registers the machine as a node in the cluster and

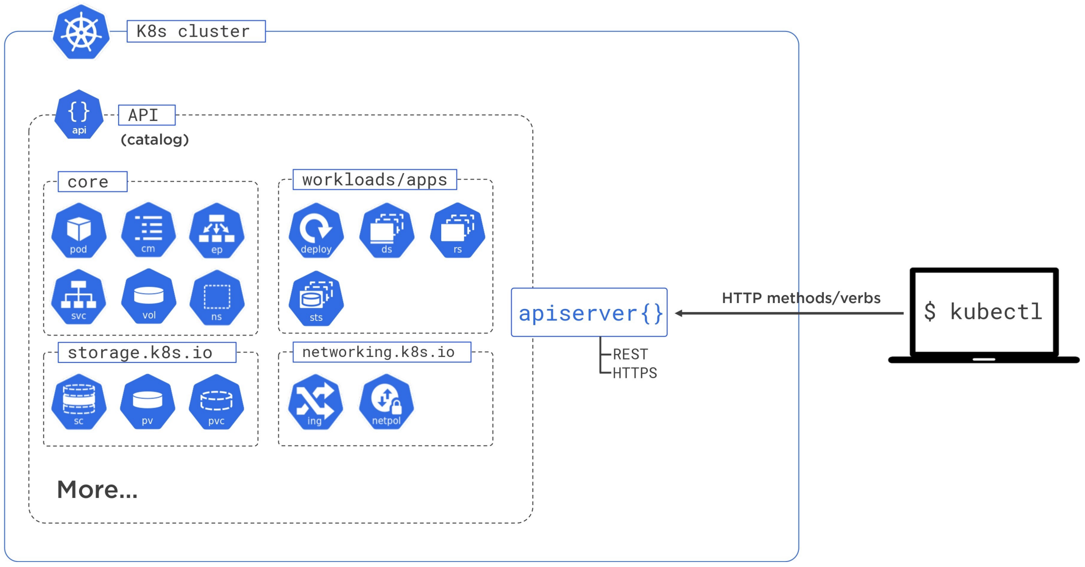
integrates its CPU, RAM, and other resources into the total cluster resource pool. Tasks in a Kubernetes cluster are executed in the form of pods, which are one or more containers bundled together as a singular deployable unit. The kubelet's role includes maintaining consistent surveillance of the API server on the master for any newly assigned pods. Upon recognizing one, it retrieves the specifications and executes the pod. Furthermore, the kubelet also maintains a reporting channel back to the API server to keep the masters updated on the state of the cluster and any running applications.
Container runtime is responsible for the execution of applications running in containers. Originally, Docker was the primary container runtime, but the component is pluggable via the Container Runtime Interface (CRI). Currently, Docker or containerd are most commonly used, but other compatible alternatives exist such as gVisor and Kata Containers.
The kube-proxy serves as the network intellect of the node, assigning unique IP addresses to each pod and performing lightweight load balancing across all pods behind a service.
In addition, some cloud services offer *nodeless Kubernetes*, where there are no nodes. This paradigm can be quite advantageous from the developer and administration perspectives, allowing one to bypass the complications of low-level infrastructure management. However, this can also be quite perplexing due to the seemingly contradictory nature of nodes being the platform where applications actually run.

2.4 Pods

In the VMware world, the atomic unit of deploying is the virtual machine; in the Docker world, it's the container and in the Kubernetes world, it's the Pod. A Pod is essentially a thin wrapper required by Kubernetes for every container and is technically defined as a **shared execution environment**. This shared environment comprises the necessary components for an application to function, such as an IP address, network port, file system, and shared memory.
The Pod is the scalable unit in Kubernetes. Scaling an application involves adding or removing Pods rather than adding more identical containers to an existing Pod. In terms of atomicity, Pod deployment is an all-or-nothing operation and is available for a service only when all its containers are running. Furthermore, once created, they cannot be restarted once they terminate: a new Pod, identical to the terminated one, is created instead of restarting the terminated Pod.

2.5 API and API Server

Kubernetes, at its core, is an intricate assembly of autonomous components that collectively provide the necessary infrastructure and features for deploying advanced cloud-native applications. Pods, Services, Deployments, SatetfullSet, and much more are all classified as objects within the Kubernetes API, which essentially encompasses all elements, including nodes. For instance, to make a pod accessible over a network or the internet, a service object is employed. Each type of object comes with a suite of features and capabilities outlined within the API. The API server,considered as a key feature of the control plane, exposes the RESTful API over HTTPS. This server operates as a web-native API that supports common methods such as POST and GET for making updates and querying states.



3 Setting up Kubernetes Cluster

3.1 Minikube

Minikube is a lightweight tool for running a single-node Kubernetes cluster locally. It is easy to install and provides a sandbox environment for learning and experimenting with Kubernetes concepts. While it offers flexibility in terms of configuration and settings, it is primarily suited for smaller workloads or individual developers. However, Minikube has limitations. It lacks scalability due to its single-node nature, making it unsuitable for production deployments requiring high availability. Additionally, resource constraints can impact the performance of larger workloads. Therefore, while Minikube is a convenient option for local development and testing, alternative approaches like Kubeadm or cloud provider managed Kubernetes services may be more appropriate for production environments with scalability and reliability requirements.
To install Minikube, first, ensure you have Kubernetes [1] and Docker [2] installed on your system. Then, follow the official Minikube installation guide [3], which provides step-by-step instructions for various operating systems.

3.2 Kubeadm

Kubeadm is a command-line utility designed to streamline the setup of robust Kubernetes clusters for production environments. By automating manual configuration steps, Kubeadm simplifies the process of initializing and managing multi-node clusters. It offers greater flexibility and control compared to Minikube, making it a popular choice for

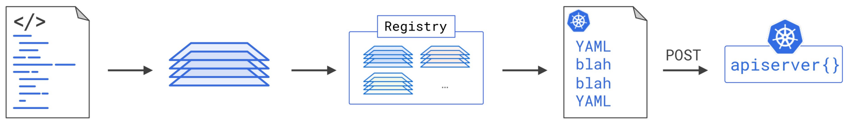
Linux-based systems. It provides fine-grained control over cluster configuration and customization, empowering administrators to tailor the cluster to their specific requirements. Moreover, Kubeadm offers enhanced flexibility for integrating with existing infrastructure and networking solutions, making it suitable for both development/testing environments and production deployments. However, it requires more manual configuration and expertise in Kubernetes concepts.

3.3 Cloud Provider Managed Kubernetes

Cloud providers like AWS, GCP, and Azure offer managed Kubernetes services such as Amazon EKS, GCP GKE, and Azure AKS. These services handle the underlying infrastructure management, simplifying the deployment and operation of Kubernetes clusters. Managed Kubernetes services provide several advantages. They abstract away the complexities of infrastructure management, allowing users to focus on deploying applications. They offer built-in scalability, high availability, and automated upgrades, ensuring clusters can grow and adapt to workload demands. Furthermore, they provide integrations with other cloud services and tools specific to the provider, enabling seamless integration within the cloud ecosystem. Managed Kubernetes services are well-suited for production workloads and enterprise-grade deployments, providing the necessary features and stability.

4 Kubernetes Pods

4.1 Deployment Workflow



The procedure for constructing and deploying an application to Kubernetes can be articulated as follows: The process begins with the application code, which is subsequently built into a container image. This container image is then stored in a designated repository. Following this, it is defined in a Kubernetes manifest, which is subsequently posted to the API server. Upon the completion of these steps, Kubernetes assumes responsibility for the remaining aspects of the process.

4.2 Deploying a Pod

In Kubernetes, a Pod is the smallest and simplest unit in the Kubernetes object model that you create or deploy. The Pod manifest is a set of instructions in a YAML or JSON format file that Kubernetes uses to create and manage a Pod.
Here's a basic example of a Kubernetes Pod object definition and a manifest file:

Listing 1: pod.yml

```yaml
#a Pod with self-build image
apiVersion: v1
kind: Pod
metadata:
  name: poc-pod
  labels:
    app: web
spec:
  containers:
    - name: poc-k8s-container
      image: skanos/poc-k8s-explorer:1.0
      ports:
        - containerPort: 3000
```

The main parts are:

– apiVersion: This is the version of the Kubernetes API you're using to create this object. The apiVersion has the format of <group>/<version>. For example, the apps group includes resources like Deployment, StatefulSet, DaemonSet, and others, but for some of the core resources like Pod, Service, Volume, e.g., the group name is not mentioned.

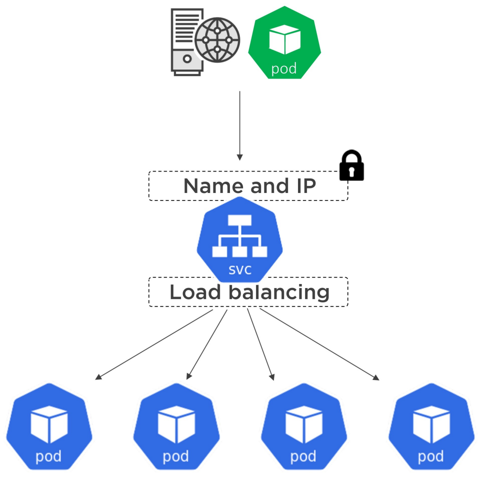
– kind: This is the kind of object you want to create. In this case, you're creating a Pod.

– metadata: This includes data that helps uniquely identify the object, including a name string, UID, and optional namespace.

– spec: This is where you specify the desired state of the object. Here, it includes the desired set of containers to run on the Pod.

The command `kubectl apply -f pod.yml` initiates a declarative deployment process using a manifest file, specified by the name provided afterwards.
The command `kubectl get pods -watch`, provides a real-time observation of the state of all the Pods within the cluster. This allows the monitoring of the status and changes to all running Pods in an ongoing manner.

5 Kubernetes Services



A Service is a REST object within the API, analogous to Pods, Nodes, and Deployments. These services are defined in YAML files, which are subsequently posted to the API server. Let's consider a scenario with multiple Pods, already deployed and operational.
It acts as an intermediary, ensuring a robust and reliable connection, despite the ephemeral nature of the Pods it manages. There are three primary service types in Kubernetes, each catering to a distinct set of requirements.

- **ClusterIP**: This is the default service type, selected if no explicit type is provided. It allocates a stable IP address within the cluster, making the service only accessible from within the cluster itself.
- **NodePort**: Building upon the ClusterIP, the NodePort service adds a cluster-wide TCP or UDP port. This facilitates access to services from outside the cluster. It operates by exposing a certain port on each node, and any traffic that is sent to this port is forwarded to the associated service.
- **LoadBalancer**: This is the most sophisticated service type, which expands upon the functionalities of both ClusterIP and NodePort services. In addition to providing an internal IP address and exposing a port across nodes, it allows your services to be exposed to the internet via the load balancer provided by your cloud platform.

It is important to note that when using LoadBalancer or NodePort services, there is no requirement to create a ClusterIP service or any other underlying service type beforehand.

5.1 NodePort Service

5.1.1 Creating NodePort Imperatively

Suppose a pod has been deployed that operates a web server. It is then essential to create a service and the most straightforward method to achieve this involves executing the following command: `kubectl expose pod poc-pod -name=poc-k8s-container`

`-target-port=3000 -type=NodePort`. The target-port option is set to 3000, which aligns with the port that the container within the pod is designed to listen to and the type option is set to NodePort, which denotes the service's exposure at a static port on each node in the cluster. This aforementioned command employs an imperative approach, given that the kubectl expose command is directly creating the service. After the service has been successfully created, the command `kubectl get services` can be executed to display the list of services currently present within the cluster.

5.1.2 Creating NodePort Declaratively

Let's delve into an example of a Kubernetes Service manifest and its various components:

Listing 2: svc-nodeport.yml

```yaml
# a Nodeport Service targeting pods with a label app: web in their metadata
apiVersion: v1
kind: Service
metadata:
  name: poc-np
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 3000
    nodePort: 31111
    protocol: TCP
  selector:
    app: web
```

– port: This is the port on which the service will be listening within the cluster and is linked to the ClusterIP. If another application within the same cluster intends to connect using the service's name (which is registered with DNS), this is the port that will be used.

– targetPort: This refers to the port on which the application inside the container is listening.

– nodePort: This is the external port that will be mapped on every node within the cluster. Although an explicit value can be assigned, it must lie within the range of 30000 to 32767.

– protocol: We have defined it as TCP, which is the default setting. However, this could be omitted or replaced with UDP if necessary.

– selector: This involves a list of labels that need to correspond with the labels on the pod we previously deployed. To confirm this, the `kubectl get pods -show-labels` command can be used.

The service is deployed using the command `kubectl apply -f svc-nodeport.yml`, and like always, the `kubectl describe services poc-np` command is used subsequently. This allows us to obtain a well-formatted view of the parameters defined within the YAML file.

5.2 LoadBalancer Service

This step involves the creation of an internet-facing load balancer, featuring a high-performing and highly reliable public IP address, to direct traffic back to our application, which is running within a pod in our cluster. However, it's crucial to note that if the use of Docker Desktop, minikube, or similar platforms, this may not function as intended.
As before, we will use a YAML configuration for this service. This time, the service type is specified as LoadBalancer. This LoadBalancer is configured to listen on port 80 and route the incoming traffic to the application listening on port 3000. As previously, the selectors used should be aligned with those of the pods we have already created.

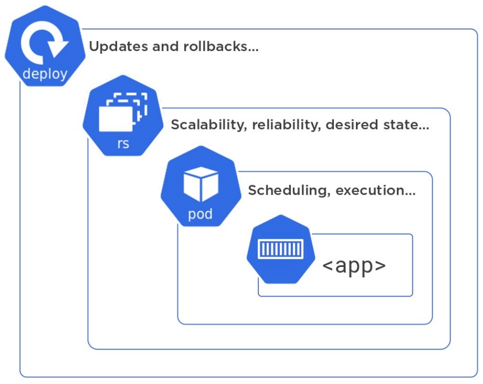
Listing 3: svc-loadbalancer.yml

```yaml
# a LoadBalancer Service targeting pods with a label app: web in their metadata
apiVersion: v1
kind: Service
metadata:
  name: poc-lb
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 3000
  selector:
    app: web
```

Upon executing the command `kubectl apply -f svc-loadbalancer.yml && kubectl get services`, our LoadBalancer service will be created and listed, along with a public IP address assigned to it.

6 Kubernetes Deployments

6.1 Replica Sets

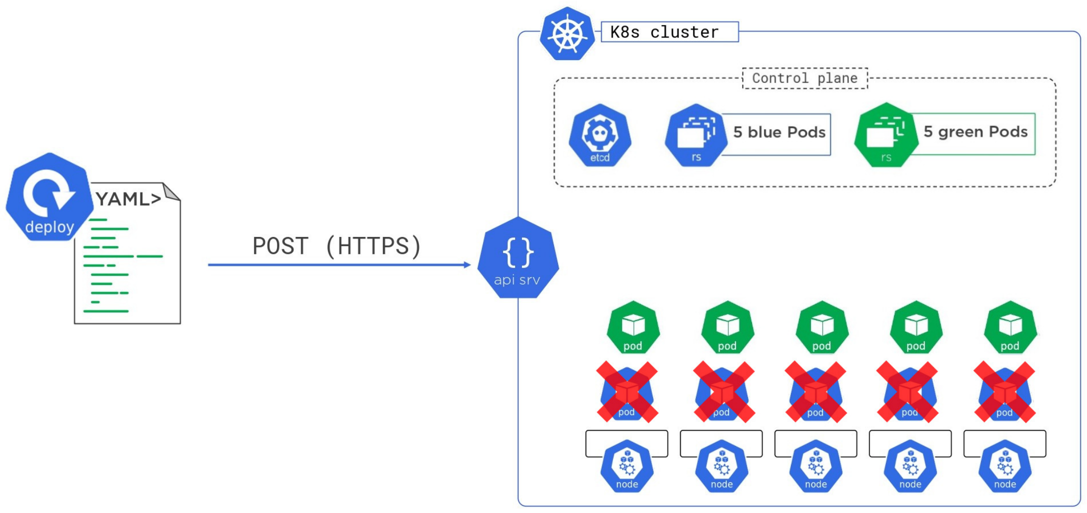


The primary responsibilities of a ReplicaSet include the maintenance of Pod replicas, handling self-healing, and managing scaling procedures. The Deployment, on the other hand, focuses on overseeing updates and managing rollbacks. ReplicaSets often remain unnoticed in the architecture, even though they play a vital role in ensuring the stability and resilience of the system.

6.2 Update and Rollback

In the Kubernetes environment, the Deployment object is crucial for managing updates and rollbacks, ensuring the desired application state. Let's walk through the process of replica management and rollbacks:

1. We commence by creating a Deployment YAML file, which outlines the desired state of the application.
2. The file is then posted as a request to the API server, where it is authenticated, authorized, and any associated policies are checked and applied.
3. The configuration is stored in the cluster store as a record of the desired state, followed by the scheduling of the requested Pods (for example, five Pods) to nodes within the cluster. Concurrently, a ReplicaSet controller operates in the background, continually ensuring the presence of the specified number of Pods (the blue ones) with the right specifications.
4. When an update, such as a new image or application update, is required, changes are made to the same Deployment YAML file.
5. This modified file is posted back to the API server.
6. In the background, Kubernetes creates a new ReplicaSet (the green ones), simultaneously scaling up the new ReplicaSet and scaling down the old one, resulting in a seamless rolling update.



It is important to note that all old ReplicaSets are retained in the background. Even though they may not be managing any Pods, their presence enables easy reversion to previous versions, making them invaluable for rollbacks. This practice facilitates the maintenance of system stability during application updates and provides a safety net in case of any undesirable outcomes.

6.3 Deployment YAML File

The K8s Deployment YAML file serves as the blueprint for how applications should be deployed and managed within a Kubernetes cluster.

Listing 4: deployment.yml

```yaml
# a deployment with 3 replicas of a self-build image
apiVersion: apps/v1
kind: Deployment
metadata:
  name: poc-deployment
  labels:
    app: web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      terminationGracePeriodSeconds: 1
      containers:
        - name: poc-k8s-container
          image: skanos/poc-k8s-explorer:1.0
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
```

An essential element is the replicas field, which dictates the number of Pods to be created and maintained in this deployment. This field is crucial for achieving desired redundancy and load balancing needs. Another key feature in this section is the label selectors (spec>selector>matchLabels). These labels play a pivotal role in identifying which Pods fall under the management of this deployment. It is crucial to ensure that these labels match the labels under the Pod template (spec>template>metadata>labels) to facilitate the correct association between the deployment and the Pods. This alignment allows the deployment to correctly manage operations such as scaling and rolling updates.

7 Comparison to Docker Swarm

7.1 Difference between Docker Swarm and K8s

The first difference between Docker Swarm and Kubernetes already starts with the installation process; Docker Swarm is integrated into Docker. If Docker is installed on a machine it can already participate or manage a Swarm. Kubernetes on the other hand has a rather lengthy install process, where many parts are optional or can be substituted. For example, the container runtime could be docker, or cotainerd, or something else entirely. There are tools to make the install process simpler, like the aforementioned Minikube, or kind.

The orchestration of a Swarm is managed through a (usually) odd number of manager nodes that decide through the RAFT algorithm a leader node that manages the swarm. If the leader is down a new one will be elected.Kubernetes has the control plane to manage a cluster. The individual parts of the control plane can be split across different nodes. The number of running instances is also mostly variable; For example, multiple kube-apiserver can be run, while running a single instance of etcd. Kubernetes, like Docker Swarm, uses the raft algorithm to ensure consensus across its master nodes. This is done through etcd, which keeps track of critical data, ensuring its correctness through the raft algorithm. The split brain condition we know from docker swarms can happen the same way in a kubernetes swarm, as they both use the same consensus algorithm.

Kubernetes mainly differentiates between two volume types in which pods can store data. Ephemeral, meaning, they will be destroyed with the pod once the pod is stopped, and persistent, meaning it will persist until it is destroyed explicitly.

7.2 Examples from Docker Swarm lecture in K8s

Setting up and configuring a production Kubernetes cluster is complex. The command line tool "kind" can be used instead to easily create a multi-node cluster for development, testing and learning purposes. kind creates nodes as docker containers from a custom image that all run on one machine. The following yml can be used to create a cluster with three control-plane nodes and three worker nodes:

Listing 5: kind-config.yml

```yaml
# a cluster with 3 control-plane nodes and 3 workers
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
- containerPort: 30000
  hostPort: 30000
- role: control-plane
- role: control-plane
- role: worker
- role: worker
- role: worker
```

The port mapping is needed because in this case the nodes are not actually separate machines or VM's but docker containers that you can't connect to by default. The config option maps the port of one of the containers to the host port using normal docker port mappings.
Now run `kind create cluster -name example -config kind-config.yml` to create cluster with the properties defined in the file and the name example. Use `kubectl get nodes -context kind-example` to get an overview over the nodes. In Docker Swarm we run a Docker image by creating a service: `docker service create -name frontend -p 8080:80 -replicas 4 nginx`. Kubernetes uses deployments for that, we can create a deployment with the same options like this: `kubectl create deployment frontend -image=docker.io/nginx:latest -replicas=4 -port=80`.
The only real difference is, that defining a port does not make the deployment reachable from the outside the cluster, only from other pods or services. There are multiple ways to make the deployment reachable from the outside. The closest to Docker Swarm's port mapping is using a NodePort service like this: `kubectl create service nodeport frontend -tcp=80:80 -node-port=30000`. The command creates a new service of type NodePort that exposes port 80 on the containers to port 80 in the cluster and to port 30000 on the nodes. With a full setup we can now reach the deployment by addressing any node on port 30000. With the kind setup the deployment is reachable at the host the kind cluster is running on on port 30000.

Scaling a deployment works very similarly to scaling a service in Docker Swarm:
Docker Swarm: `docker service scale frontend=6`
Kubernetes: `kubectl scale deployment/frontend -replicas=6`

Deploying container updates is very similar too. The parallelism options can be set in a yaml file.
Docker Swarm: `docker service update -image nginx:latest -update-parallelism 2 -update-delay 15s frontend`
Kubernetes: `kubectl set image deployments/frontend frontend=docker.io/nginx:latest`
Using a HostPath volume we can create a bind mount from the containers to the node host. This is only possible in yaml. The following yaml creates the same deployment as the commands above and adds a HostPath volume that maps /data on the node hosts to /usr/share/nginx/html in the containers:

Listing 6: frontend-deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: data-volume
      volumes:
      - name: data-volume
        hostPath:
          path: /data
          type: Directory
```

Using `kubectl apply -f frontend-deployment.yml` we can apply this config all at once instead of using multiple commands. This does not include the service yet, which can be described in another yaml file and applied in the same way.

References

[1] https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/

[2] https://docs.docker.com/get-docker/

[3] https://minikube.sigs.k8s.io/docs/start/

[*] All images retrieved from the course "Getting Started with Kubernetes" by Nigel Poulton