**Kubernetes** is a tool that helps us to run and manage applications in containers. It was developed by Google Lab in 2014, and it is also known as k8s. It is an open-source container orchestration platform that automates the deployment, management, and scaling of container-based applications in different kinds of environments like physical, virtual, and cloud-native computing foundations. Containers are isolated from each other so that multiple containers can run on the same machine without interrupting anyone else. It allows us to deploy and manage container-based applications across a Kubernetes cluster of machines.



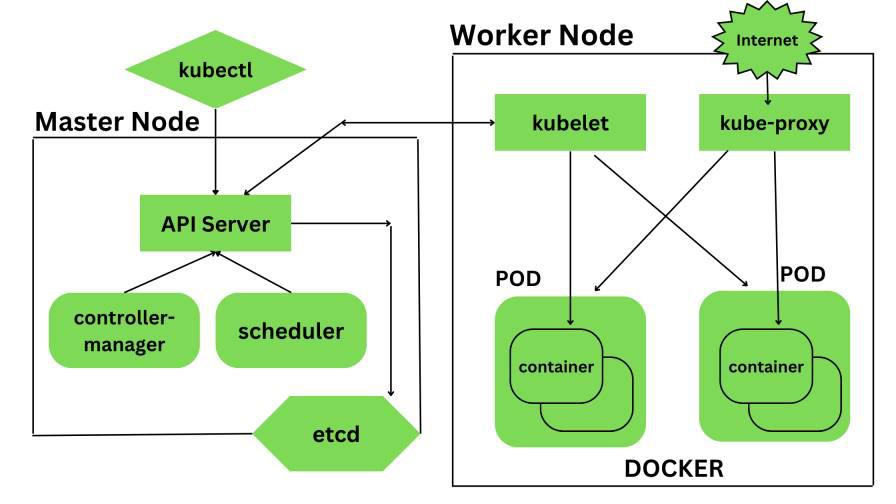
Kubernetes is an open-source platform that manages Docker containers in the form of a cluster. Along with the automated deployment and scaling of containers, it provides healing by automatically restarting failed containers and rescheduling them when their hosts die. This capability improves the application’s availability.

**Features of Kubernetes:**

1. **Automated Scheduling**– Kubernetes provides an advanced scheduler to launch containers on cluster nodes. It performs resource optimization.
2. **Self-Healing Capabilities**– It provides rescheduling, replacing, and restarting the containers which are dead.
3. **Automated Rollouts and Rollbacks**– It supports rollouts and rollbacks for the desired state of the containerized application.
4. **Horizontal Scaling and Load Balancing**– Kubernetes can scale up and scale down the application as per the requirements.
5. **Resource Utilization**– Kubernetes provides resource utilization monitoring and optimization, ensuring containers are using their resources efficiently.
6. **Support for multiple clouds and hybrid clouds**– Kubernetes can be deployed on different cloud platforms and run containerized applications across multiple clouds.
7. **Extensibility**– Kubernetes is very extensible and can be extended with custom plugins and controllers.
8. **Community Support-** Kubernetes has a large and active community with frequent updates, bug fixes, and new features being added.

## Architecture of Kubernetes

Kubernetes follows the client-server architecture where we have the master installed on one machine and the node on separate Linux machines. It follows the master-slave model, which uses a master to manage Docker containers across multiple Kubernetes nodes. A master and its controlled nodes(worker nodes) constitute a **“Kubernetes cluster”**. A developer can deploy an application in the docker containers with the assistance of the Kubernetes master.



*Architecture of Kubernetes*

### 1. Kubernetes- Master Node Components –

Kubernetes master is responsible for managing the entire cluster, coordinates all activities inside the cluster, and communicates with the worker nodes to keep the Kubernetes and your application running. This is the entry point of all administrative tasks. When we install Kubernetes on our system we have four primary components of Kubernetes Master that will get installed. The components of the Kubernetes Master node are:

**a.) API Server**– The API server is the entry point for all the REST commands used to control the cluster. All the administrative tasks are done by the API server within the master node. If we want to create, delete, update or display in Kubernetes object it has to go through this API server.API server validates and configures the API objects such as ports, services, replication, controllers, and deployments and it is responsible for exposing APIs for every operation. We can interact with these APIs using a tool called **kubectl**. *‘kubectl’ is a very tiny go language binary that basically talks to the API server to perform any operations that we issue from the command line. It is a command-line interface for running commands against Kubernetes clusters*

**b.) Scheduler**– It is a service in the master responsible for distributing the workload. It is responsible for tracking the utilization of the working load of each worker node and then placing the workload on which resources are available and can accept the workload. The scheduler is responsible for scheduling pods across available nodes depending on the constraints you mention in the configuration file it schedules these pods accordingly. The scheduler is responsible for workload utilization and allocating the pod to the new node.

**c.) Controller Manager**– Also known as controllers. It is a daemon that runs in a non terminating loop and is responsible for collecting and sending information to the API server. It regulates the Kubernetes cluster by performing lifestyle functions such as namespace creation and lifecycle event garbage collections, terminated pod garbage collection, cascading deleted garbage collection, node garbage collection, and many more. Basically, the controller watches the desired state of the cluster if the current state of the cluster does not meet the desired state then the control loop takes the corrective steps to make sure that the current state is the same as that of the desired state. The key controllers are the replication controller, endpoint controller, namespace controller, and service account, controller. So in this way controllers are responsible for the overall health of the entire cluster by ensuring that nodes are up and running all the time and correct pods are running as mentioned in the specs file.

**d.) etcd**– It is a distributed key-value lightweight database. In Kubernetes, it is a central database for storing the current cluster state at any point in time and is also used to store the configuration details such as subnets, config maps, etc. It is written in the Go programming language.

### 2. Kubernetes- Worker Node Components –

Kubernetes Worker node contains all the necessary services to manage the networking between the containers, communicate with the master node, and assign resources to the containers scheduled. The components of the Kubernetes Worker node are:

**a.) Kubelet**– It is a primary node agent which communicates with the master node and executes on each worker node inside the cluster. It gets the pod specifications through the API server and executes the container associated with the pods and ensures that the containers described in the pods are running and healthy. If kubelet notices any issues with the pods running on the worker nodes then it tries to restart the pod on the same node. If the issue is with the worker node itself then the Kubernetes master node detects the node failure and decides to recreate the pods on the other healthy node.

**b.) Kube-Proxy**– It is the core networking component inside the Kubernetes cluster. It is responsible for maintaining the entire network configuration. Kube-Proxy maintains the distributed network across all the nodes, pods, and containers and exposes the services across the outside world. It acts as a network proxy and load balancer for a service on a single worker node and manages the network routing for TCP and UDP packets. It listens to the API server for each service endpoint creation and deletion so for each service endpoint it sets up the route so that you can reach it.

**c.) Pods**– A pod is a group of containers that are deployed together on the same host. With the help of pods, we can deploy multiple dependent containers together so it acts as a wrapper around these containers so we can interact and manage these containers primarily through pods.

**d.) Docker**– Docker is the containerization platform that is used to package your application and all its dependencies together in the form of containers to make sure that your application works seamlessly in any environment which can be development or test or production. Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Docker is the world’s leading software container platform.

A diagram of a computer network

Description automatically generated

### Application of Kubernetes

* Microservices architecture: Kubernetes is well-suited for managing microservices architectures, which involve breaking down complex applications into smaller, modular components that can be independently deployed and managed.
* Cloud-native development: Kubernetes is a key component of cloud-native development, which involves building applications that are designed to run on cloud infrastructure and take advantage of the scalability, flexibility, and resilience of the cloud.
* Continuous integration and delivery: Kubernetes integrates well with CI/CD pipelines, making it easier to automate the deployment process and roll out new versions of your application with minimal downtime.
* Hybrid and multi-cloud deployments: Kubernetes provides a consistent deployment and management experience across different cloud providers, on-premise data centers, and even developer laptops, making it easier to build and manage hybrid and multi-cloud deployments.
* High-performance computing: Kubernetes can be used to manage high-performance computing workloads, such as scientific simulations, machine learning, and big data processing.
* Edge computing: Kubernetes is also being used in edge computing applications, where it can be used to manage containerized applications running on edge devices such as IoT devices or network appliances.

## Kubernetes Basic Concepts

Now we know broadly what Kubernetes is and a little bit about its most important partner Docker, but what does Kubernetes look like once you jump into it?

Below we’ll break down the basic components present in any Kubernetes app and show how they build on each other. We’ll also look at some of the additional tools sure to help you along the way.

### Pods

These are the smallest unit of application in Kubernetes. Pods represent a single, isolated instance of an application and the resources needed to execute it, each having their own IP address.

Pods are made up of one or more containers that work together and share a life cycle on the same node; each pod could be composed of a single container or multiple containers, depending on its complexity.

This can be advantageous as all containers within the same pod can communicate without the need for additional setup from the user.

These pods, however, are highly isolated and cannot communicate with other pods. This is where our next component, services, comes into the picture.

### Services

Services are an abstraction that sit one level above pods, acting as a director between individual pods and the outside world.

Services define a policy to access selected pods. Once this is done, pods in this service’s set can communicate freely until the policy to access is changed.

Services are also convenient when creating complex, interdependent programs as they set a single Domain Name Service (DNS) record for all pods within their service. Another part of the program, a deployment (see below), can then use this DNS name to access information from these pods without needing to know the IP addresses for each.

In other words, another part of the program can connect to this pod group through the shared DNS record without needing to track status or information of each pod.

Group based access allows for greater flexibility than individual access as pods can be added or removed from the service group without code revision.

### Deployments

Our final component of Kubernetes, Deployments, takes the hassle out of upgrading pods. Normally, when upgrading a system, one would have to shut off all old instances then reboot the system using the upgraded instances, resulting in a period of downtime.

Kubernetes’ deployments allow us to sidestep this problem by allowing for “rolling updates” during which pods are taken down one-by-one, upgraded to the new version, and verified as functional before moving onto the next.

In doing so, Kubernetes ensures that not only is there no downtime but also that each pod is upgraded as it should. This process can also be repeated in the opposite order should the new program contain an error, resulting in an automated rollback to the previous version.

For a closer look at pods, services, deployments, and replication controllers check out our [previous article](https://www.educative.io/blog/pods-services-deployments).

### Minikube

The first of our additional tools, Minikube creates a virtual testing ground for all your containerized programs. When run, Minikube creates a Virtual Machine (VM) on your computer which will simulate the behavior of a physical system without the risk of making unwanted changes to your machine.

This VM is a simple single-node cluster, meaning that it behaves like a group of computers with only one machine hooked up. This simplified cluster form and simulated behavior make Minikube the perfect development and testing environment for unfinished programs or for simply [learning container-based programming](https://www.educative.io/catalog/kubernetes).

While this is ideal for Linux and Mac users, Windows users may have a hard time getting it started. For Windows, instead, try Docker Desktop; it has the same test environment functionality with a simpler fit into Windows’ infrastructure.

### kubectl

For our next tool, we have kubectl, a command-line application to manage Kubernetes clusters. In many ways, the kubectl command is its own coding language with exclusive syntax and complexity. While difficult to pick up due to its extensiveness, kubectl offers unmatched control to developers, making it the most popular cluster manipulation tool for Kubernetes out there right now.

Combined with Minikube, these two tools are essential for those seeking to dive into Kubernetes.

### Kubernetes Cluster

Clusters are groups of servers all working together as a system toward the same goal. These servers are commonly referred to as nodes and can each work on independent tasks which culminate to accomplishing the system’s goal.

Each cluster has a list of conditions defined in its config files which the cluster expects in order to run correctly, known as a “desired state”. This state could feature specifications such as which workloads should be running, container images the cluster will need access to, or hardware resources that should be available on a given machine.

### Master and worker nodes

Nodes in a cluster have different functions; some are masters and others are workers. In each cluster, there is a master node and at least one worker node.

The master node is charged with maintaining the cluster’s desired state, freeing up resources, checking the status of each condition, and so on.

It also manages the scheduling of pods across nodes in a cluster, distributing jobs to ensure all nodes are working while not becoming overloaded. The master node has various components like API Server, Controller Manager, Scheduler, and ETCD.

Worker nodes are responsible for everything else involved in running a program and do most of the lifting. These nodes can have multiple pods running on them, working on each as assigned by the master nodes. As worker nodes are mostly isolated, clusters can be scaled with more worker nodes without issue.

## How to create a Kubernetes Cluster

To begin, we’ll create our cluster. Thanks to our tools, Minikube or Docker Desktop, this is simple. This will be the only step where the inputs deviate based on the tool.

Minikube users, enter:

minikube start

For Docker Desktop, your cluster has begun automatically.

Now to test that our cluster was made correctly, enter:

kubectl get svc

The following table will print. Yours may look slightly different, but the two leftmost fields, Kubernetes and clusterIP, will be the same.

NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE  
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   6s

This shows us that the default service, Kubernetes, is currently running. As this service is always initialized at cluster startup, we know the cluster is running as expected!

## How to deploy a Kubernetes app

Now we’ll create our first deployment. For this, we’ll be pulling a test image from the Google Container Registry (GCR) called hello-node. Creating this deployment will come with one pod built-in.

To do this, paste the following line:

kubectl create deployment hello-node --image=gcr.io/hello-minikube-zero-install/hello-node

Now, to check that this deployment is running correctly, enter:

kubectl get deployments

NAME         READY   UP-TO-DATE   AVAILABLE   AGE  
hello-node   1/1     1            1           170m

This will print a list of the deployments currently running on this cluster. As we’ve only created the one, you’ll only see hello-node listed. Congratulations, you’ve deployed your app!

## Exploring your Kubernetes app

While we’ve got it, might as well look around! We’ll use the get command to see that all the parts we learned about above are now present and working on our app!

First, let’s look at our nodes, enter:

kubectl get nodes

NAME             STATUS   ROLES    AGE    VERSION  
docker-desktop   Ready    master   1d2h   v1.15.5

This will list the names, status, and roles of all the nodes in our cluster. Since our cluster is just a single device test environment, there will be only one.

Now we’ll look at our pods, enter:

kubectl get pods

Again, a table will print with some useful information.

NAME                          READY   STATUS    RESTARTS   AGE  
hello-node-55b49fb9f8-wwpwr   1/1     Running   0          36s

Here, it’s important to note that our pod has only one instance, shown by the 1/1 in the ready column. We can also see that it is currently running from the status column, letting us know that it has not failed.

As a result, the get pods command is useful when troubleshooting a more complex system as it allows you to see which of your many pods is malfunctioning.

If you need more information, you can also use the describe command for specific pods, nodes or even deployments to see a plethora of information available on each resource.

Let’s try it for our deployment, enter:

kubectl describe deployment hello-node

Name:                   hello-node  
Namespace:              default  
CreationTimestamp:      Mon, 01 Jan 2020 19:26:58 -0700  
Labels:                 app=hello-node  
Annotations:            deployment.kubernetes.io/revision: 1  
Selector:               app=hello-node  
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable  
StrategyType:           RollingUpdate  
MinReadySeconds:        0  
RollingUpdateStrategy:  25% max unavailable, 25% max surge  
Pod Template:  
  Labels:  app=hello-node  
  Containers:  
   hello-node:  
    Image:        gcr.io/hello-minikube-zero-install/hello-node  
    Port:         <none>  
    Host Port:    <none>  
    Environment:  <none>  
    Mounts:       <none>  
  Volumes:        <none>  
Conditions:  
  Type           Status  Reason  
  ----           ------  ------  
  Available      True    MinimumReplicasAvailable  
  Progressing    True    NewReplicaSetAvailable  
OldReplicaSets:  <none>  
NewReplicaSet:   hello-node-55b49fb9f8 (1/1 replicas created)  
Events:  
  Type    Reason             Age   From                   Message  
  ----    ------             ----  ----                   -------  
  Normal  ScalingReplicaSet  95s   deployment-controller  Scaled up replica set hello-node-55b49fb9f8 to 1

You’ll not recognize most of this information if you’re just starting out, however for the experienced Kubernetes user, this ready access to information is one of the aspects which makes Kubernetes so appealing. With both get and describe commands, all the information needed to troubleshoot a failure is just a command-line away!

## How to expose your Kubernetes app

Up until now, our app has been fully isolated in our test environment. With real programs, you’ll almost always need to have your app send web requests to other outside web apps.

To do this, we’ll use the expose command which will create a new service instance with the same name as our deployment and will automatically define the port configuration to allow a connection. As part of the command, we define which port the service should listen on. In this case, we’ll use port 8080.

To try this yourself, enter:

kubectl expose deployment hello-node --type=LoadBalancer --port=8080

To see this service in action, we can once again enter:

kubectl get svc

NAME       TYPE         CLUSTER-IP     EXTERNAL-IP  PORT(S)         AGE  
hello-node LoadBalancer 10.108.188.234 localhost    8080:32505/TCP  7s  
kubernetes ClusterIP    10.96.0.1      <none>       443/TCP         1h

Here we see that where we once had only one service, we now have two, the original kubernetes and new service hello-node. Notice how the latter differs in both the EXTERNAL-IP and the PORT(S) field due to it being public. NodePorts are the published IP addresses for external users to access the services.

Now that our app is exposed, we can access the web application running inside the hello-node pod to print a message.

To do this, enter:

 curl http:///localhost:8080

Hello World!

And like that, we’ve called an external IP to execute our pod and had it print!

## How to scale a Kubernetes app

While our app now works fully and is reachable by external sources, what happens if our pod fails? Or what happens if the number of requests reaching our program suddenly spikes?

To protect against these cases, we need to scale up our app; creating more instances of our hello-node pod to delineate requests to or to step in if a pod fails. The best part is, with Kubernetes this takes but a single command to create and will do the above jobs automatically from then on!

To scale up our app, enter:

kubectl scale --replicas=3 deployment/hello-node

This will set our program to maintain a state of three running instances of the hello-node pod rather than just one.

To check this, enter:

kubectl get deployment hello-node

This should print a screen like so:

NAME         READY   UP-TO-DATE   AVAILABLE   AGE  
hello-node   3/3     3            3           135m

Notice the three pods under this deployment listed in the ready column.

Or to see the pods independently you can enter:

kubectl get pods

NAME                          READY   STATUS    RESTARTS   AGE  
hello-node-55b49fb9f8-fxjnj   1/1     Running   0          2m34s  
hello-node-55b49fb9f8-jlfwq   1/1     Running   0          2m34s  
hello-node-55b49fb9f8-zhf9f   1/1     Running   0          136m

This once again shows us that where there was once only a single pod, we now have three separate but identical instances of the hello-node pod.

## How to update a Kubernetes app

Finally, we know that no app rules forever; eventually, a new version will replace the old. In our case, say a new version of our hello-node pod is created and we want to use it to replace all of our old pods. Well as we discussed previously, Kubernetes has us covered as we can make this upgrade with no downtime using a rolling update.

As there is no new version of hello-node, this would give an error message if entered.

However, if there were an updated version, you would apply it by entering:

kubectl set image deployments/hello-node hello-node=myContainers/hello-node:v2

To break this down, we first mark that we’d like to set a new image, then specify the type we’re going to adapt, deployments, then which deployment, hello-node. After the first mention of our deployment, we’ve told the command what we’d like to edit, at which point we then provide the new image, myContainers/hello-node:v2.

Kubernetes will then take each of our old pods down one by one and replace them with our upgraded version. This takes a moment and does get longer depending on the number of pods which must be upgraded. To check if the upgrade rollout is successful, we would enter:

 kubectl rollout status deployments/hello-node

Which, if completed, would return:

 deployment "hello-node" successfully rolled out

With that, we’ve finished our creation and exploration of a Kubernetes basic program. To conclude your masterpiece, now simply enter the two lines:

kubectl delete service hello-node  
kubectl delete deployment hello-node

This will clean up your system and leave your system ready for your next Kubernetes project!