**Kubernetes – Part 3**

**Deployments:**

For an instance let’s forget about PODs, replicasets and other methods, just think about how we might want to deploy our application in the production environment; say for ex: we have a web server that needs to be deployed in a production environment, and we need many web server instances for various reasons; secondly, whenever newer versions of application build become available on the docker registry, you would like to upgrade your docker instances seamlessly, However, when you upgrade your instances you do not want to upgrade all of them at once as we just did, this may impact users accessing our applications so we might want to upgrade them one after the other and that kind of upgrade is known as “Rolling updates” suppose one of the upgrades you performed resulted in an unexpected error and you’re asked to undo the recent change, you would like to be able to roll back the changes that were recently carried out. Finally say for example you would like to make multiple changes to your environment such as upgrading the underlying web server versions as well as scaling your environment and also modifying the resource allocations etc., you do not want to apply each change immediately after the command is ruin, instead you like to apply a pause to your environment make the changes and then resumes so that all the changes are rolled out together. All of these capabilities are available with the kubernetes deployments. We discussed about pods which deploys single instances of our application such as the web application, each container is encapsulated in pods. Multiple such pods are deployed using replication controllers or replica sets and then comes deployment, which is a Kubernetes object that comes higher in the hierarchy, the deployment provides us with the capability to upgrade the underlying instances seamlessly using rolling updates, undo changes and pause and resume changes as required.

***Steps:***

1. Create a file (vi deployment-definition.yml)

apiVersion: apps/v1

kind: Deployment

metadata:

name: myapp-deployment

labels:

app: myapp

type: front-end

spec:

template:

metadata:

name: myapp-pod

labels:

app: myapp

type: front-end

spec:

containers:

- name: nginx-container

image: nginx

replicas: 3

selector:

matchLabels:

type: front-end

1. Create the file (kubectl create –f deployment-definition.yml)
2. To get the list, give (kubectl get deployments) [as we mentioned here with replicas we can see here the replicas has been created along with them]
3. To confirm the same, call (kubectl get replicaset) [we will be able to see a new replica set in the name of the deployment]
4. The replica sets ultimately creates the Pod, so we call (kubectl get pods) we will be able to see the pods with the name of the deployment.

**Deployment Updates, Rollbacks:**

***Rollout and Versioning:***

Before we look at how we upgrade out application, let’s try to understand rollouts and versioning in a deployment.

When you 1st create a deployment it triggers a rollout, a new rollout creates a new deployment revision. Let’s call it as (Revision 1)

In the future when the application is upgraded, meaning when the container version is updated to a new one, a new rollout is triggered and a new deployment revision is created (Revision 2)

This helps us to keep track of the changes made to out deployment and enables us to roll back to a previous version of deployment if necessary.

***We can see the status of Rollout by running the command: kubectl rollout status deployment/myapp-deployment***

***To see the revisions and history of rollout, run the command: kubectl rollout history deployment/myapp-deployment***

**Deployment Strategy:**

There are 2 types of deployment strategies,

Option 1: let’s say we have 5 replicas of web application instance deployed, to upgrade to a newer version is to destroy all of these and then create newer versions of application instances. The problem with this kind of strategy, is application down which means inaccessible to users, this strategy is known as the “Recreate strategy” and thankfully this is not the default deployment strategy.

Option 2: Instead of destroying everything instead we take down the older version and bring up the newer version one by one, this way the application never goes down and the upgrade is seamless, this strategy is known as the “Rolling update” is the default deployment strategy.

How exactly do we update our deployment?

When I say update, this could be different things, such as updating your application version by updating the version of Docker containers used, updating their labels or updating the number of replicas, etc.,

For ex:

apiVersion: apps/v1

kind: Deployment

metadata:

name: myapp-deployment

labels:

app: myapp

type: front-end

spec:

template:

metadata:

name: myapp-pod

labels:

app: myapp

type: front-end

spec:

containers:

- name: nginx-container

image: **nginx:1.7.1**

replicas: 3

selector:

matchLabels:

type: front-end

Since we already have a deployment definition file, it is easy for us to modify this file once we make the necessary changes, then we need to apply the changes, by running

***kubectl apply –f deployment-definition.yml***

Rollback:

Say, for instance once you upgrade your application, you realize somethings wrong with the new version, so you want to rollback the update. Kubernetes deployment allow you to roll back to a previous revision, to undo a change, follow the command:

***kubectl rollout undo deployment/myapp-deployment***

The deployment will then destroy the pods in the new replica set and bring the older ones up in the old replicaset, means your application is back to its older format.

**Networking in Kubernetes:**

Let’s start with a single node kubernetes cluster, this node has an IP address (192.168.1.2) through this IP address only we were accessing the Kubernetes node, SSH, etc., and consider we have a single Pod, as we know a Pod hosts a container, unlike in the Docker world where an IP address is always assigned to a Docker container, In kubernetes world, the IP address is assigned to a Pod, means each Pod in the kubernetes has its own internal IP address, consider (10.244 series) and the IP assigned to the pod is (10.244.0.2) == So how this is getting this IP address? When Kubernetes is initially configured, we create an internal private network with the address (10.244.0.0) and all the ports are attached to it, when we deploy multiple ports, they all get a separate IP assigned from this network. The pods can communicate to each other through this IP, but accessing the other parts using this internal IP address may not be a good idea as it’s subject to change, when pods are recreated, we will see better ways to establish communication between pods in a while, for now it’s important to understand how the internal networking works in kubernetes.

What if we have a multi-node cluster setup?

Consider we have 2 nodes with an IP address



Note that they are not part of the cluster yet, each of them has a single pod deployed, these pods are attached to an internal network and they have their own IP addresses assigned. However if you look at the internal network addresses, you can see they are same as well as the paths deployed have the same address. These are not going to work well, when the nodes are part of the same cluster, the pods have the same IP addresses assigned to them and that leads to IP conflicts in the network.



When a kubernetes cluster is set up, kubernetes does not automatically set up any kind of networking to handle these issues. As a matter of fact, kubernetes expects us to set up the networking to meet certain fundamental requirements – some of these are

1. All containers/PODs can communicate to one another without configure NAT
2. All nodes must be able to communicate with all containers and vice-versa without NAT.

Kubernetes expects us to setup a networking solution that meets these criteria. Fortunately, we don’t have to set it up all on our own as there are multiple pre-built solutions available. Some of them are the Cisco ACI networks psyllium, Big cloud, fabric, flannel, VMware NSXt and calico, depending on the platform you’re deploying your kubernetes cluster on, we need choose one accordingly. For ex: if you were setting up a kubernetes cluster from scratch on our own systems, we may choose something like calico or flannel. If we were deploying on a VMware environment NSXt may be a good option.

So according to our setup we may use custom networking either with flannel or calico, this will manages the network and IPs ion my nodes and assigns a different network address for each network in the node,



This creates a virtual network of all pods and nodes where they are all assigned a unique IP address, and by using simple routing techniques, the cluster networking enables communication between the different pods or nodes to meet the networking requirements of kubernetes, thus all pods now can communicate to each other using the assigned IP address.

**Services:**

Kubernetes services enable communication between various components within and outside of the application. This helps us connect applications together with other applications or users. For ex: our application has groups of pods running various sections, such as a group for serving a frontend load to users and other group for running backend processes, and the 3rd connecting to and external data source. It is services that enables connectivity between these groups of pods. Thus, services enable loose coupling between micro services in our application.

***Use Case:***

We deployed our pod having a web application running on it, how do we, as an external user access the web page? Here the node is running on IP address, (192.168.1.2) my laptop is on the same network as well (192.168.1.10), the internal pod network is in the range of (10.244.0.0) and the pod has an IP (10.244.0.2), we won’t be able to ping here as it is running on a separate network, So what are the options to see the webpage?



1st, if we ssh into the kubernetes node at (192.168.1.2) from the node we would be able to access the pod’s webpage by doing a (curl httpd://10.244.0.2), if the node has GUI, we can get the output through browser following the address (<http://10.244.0.2>) but this is from inside the kubernetes node, and this is not what I really want, I want to be able to access the web server from my own laptop without having to ssh into the node and simply by accessing the IP of the kubernetes node. So, we need something in the middle to help us map requests to the node from our laptop through the node to the pod running the web container, this is where the Kubernetes service comes into play. The kubernetes service is an object, just like pods, replicasets, and deployments. One if its use case is to listen to a port on the node and forward requests on that pod where the application is running. This type of service is knows as a ***node port*** service because the service listens to a port on the node and forward requests to the pod.

**Types of Services:**

***Node Port:*** where the service makes an internal pod accessible on the port on the node.



***ClusterIP:*** The service creates a virtual IP inside the cluster to enable communication between different services, such as a set of frontend servers to a set of backend servers.



***LoadBalancer:*** where it provisions a load balancer for our application in supported cloud providers. [ex: distribute load across the different web servers in your frontend tier]



**Service – NodePort**



There are 3 ports involved, the port on the pod where the actual web server is running is 80 and this is referred to the Target port, because that is where the service forwards the requests. The 2nd port is the port on the service itself, and it is simply referred as port, [remember these terms are from the view point of the service], the services is in fact like a virtual server inside the node, and the service has the own IP address inside the cluster, and that IP address is called the clusterIP of the service. And finally we have the port on the node itself, which we use to access the web server externally and that is known as the NodePort, here it is set to (30008), NodePort do have a range by default (30000 – 32767).

***Steps:***

1. Create a file (vi service-definition.yml)

apiVersion: v1

kind: Service

metadata:

name: myapp-service

spec:

type: NodePort

ports:

* targetPort: 80

port: 80

nodePort: 30008

selector:

app: myapp

type: front-end

1. Create the file (kubectl create –f service-definition.yml)
2. To see the service, run (kubectl get services)

***Tasks:***

Access your service via browser

As we know in real-time we are going to have multiple pods, in the production environment, we have multiple instances of web application running for high availability and load balancing purposes. We have here the similar pods running out web application, they all have the same labels (myapp) and the same label used as a selector during the creation of service. So when a service is created, it looks for the matching pod with the label and find 3 of them. The service then automatically selects all the three pods as endpoints to forward the external requests, we don’t want to do anything additional for this one. And if we are checking what algorithm, it uses to balance the load across the 3 different pods, this is where the service acts as a built-in loadbalancer to distribute load across different pods.

**ClusterIP:**

A full stack web application typically has different kinds of pods hosting different parts of an Application, like we have multiple pods for frontend and backend and we do have few for our database, as we know in; this application the frontend services needs to communicate with the backend server and the backend needs to connects with the database; so what way we can connect all these? We already know each pod is going to have their IP address allocated, and these IPs are not a static one, hence we cannot rely on these IP addresses for internal communication as this is not a static one, in another scenario as we have multiple pods for each part of the application which pod it will get connected from front-end to backend and to database how it is getting connected? There comes the kubernetes service that helps us group the pods together and provide a single interface to access the pods in a group, this enables us to easily and effectively deploy a Microservices based application on kubernetes cluster and it is known as the ClusterIP.



***Steps:***

1. Create a file (vi service-definition.yml)

apiVersion: v1

kind: Service

metadata:

name: back-end

spec:

type: ClusterIP

ports:

* targetPort: 80

port: 80

selector:

app: myapp

type: back-end

1. Create the file (kubectl create –f service-definition.yml)
2. To see the service, run (kubectl get services)

**LoadBalancer:**

This supports only for specific cloud platforms, and mostly we are going to use this to manage the traffic with providing with a specific URL, in real-time as we know our applications are going to run in multiple pods, and it is possible for us to provide all that IPs to customers to access our application, and even these are not static as well, so how can a customer connect with our application? Via URL? And these can be set by the native cloud providers on their own environment, so creating a cluster and attaching that nodes with the Load balancer is going to do that job appropriately, [this method is rarely used, and we can use it when we are creating clusters directly using cloud service providers]

Reference Link for Amazon EKS: <https://docs.aws.amazon.com/eks/latest/userguide/create-cluster.html>

**Storage:**

Just like Docker, the pods created in kubernetes are transient in nature, when a pod is created to process data and then deleted the data process by it, gets deleted as well. For this, we attach a volume to the Pod, the data generated by the pod is not stored in the volume and even after the pod is deleted the data remains.

**Volumes & Mounts:**

***Use case:***

We have a single kubernetes cluster, we create a simple pod that generates a random number between 1 & 100 and writes that to a file at /opt/no.out; it then gets deleted along with the random number, to retain the number generated by the pod we create a volume, and this volume comes along with the storage; when we create the volume itself we can configure this storage in different ways; let’s consider for time-being we will use a directory on the host; here we can specify the path (/data) on the host, this way any files created in the volume would be stored in the directory on my node. Once the volume is created, to access it from a container we mount the volume to a directory inside the container, we use the volume mounts filed in each container to mount the data volume to the directory, /opt within the container; after this whenever a random number written to /opt mount inside the container which happens to be on the data volume, which is in fact the data directory on the host. Here after even the pod gets deleted, the file with the random number still lives on the host.



**Persistent Volumes:**

We normally configure volumes in the (pod-definition file), so every configuration information required to configure storage for the volume goes within the pod definition file, [consider we have a large environment with a lot of users deploying a lot of pods, the users would have to configure storage every time for each pod, whatever storage solutions is used the users who deploys the pods would have to configure that on all pod definition files in his environment. Everything a changes to be made the user would have to make them on all of his pods, instead, you would like to manage storage more centrally where you would like to configured in a way that an administrator can create a large pool of storage and then have users carve out pieces from it as required] that is where persistent volumes can help us.

Persistent volume is a cluster-wide pool of storage volumes configured by an administrator to be used by users deploying application on the clusters. The users can now select storage from this pool using Persistent Volume Claims (PVC),

***Steps:***

1. Create a file (vi pv-definition.yaml)

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv-volume

spec:

accessModes:

- ReadWriteOnce

capacity:

storage: 1Gi

hostPath:

path: /tmp/data

1. Create the file (kubectl create –f pv-definition.yml)
2. To see the volume, run (kubectl get persistentvolume)

Access Modes defines how a volume should be mounted on the hosts, whether only ReadOnly mode, or ReadWrite mode etc., (supported values are: ReadOnlyMany, ReadWriteOnce, ReadWriteMany)

**Persistent Volume Claims:**

PV & PVC are 2 separate objects in the Kubernetes namespace, An administrator creates a set of persistent volumes, and a user create persistent volume claims to use the storage, Once that persistent volume claims are created kubernetes binds the persistent volumes to claims based on based on the request and properties set on the volume, every PVC is bound to a single Persistent volume, during the binding process, kubernetes tries to find a Persistent volume that has sufficient capacity as requested by the claim and any other request properties such as access modes, volume modes, storage classes etc.,

***Steps:***

1. Create a file (vi pvc-definition.yaml)

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: myclaim

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 500Mi

1. Create the file (kubectl create –f pvc-definition.yml)
2. To see the volume, run (kubectl get persistentvolumeclaim)
3. To get the overall information, run (kubectl describe pvc myclaim)

**Difference between Persistent Volume and Persistent Volume Claims:**

In Kubernetes, Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) are two fundamental concepts related to storage management within the cluster. Here's the difference between the two:

***Persistent Volume (PV):***

A Persistent Volume is a piece of storage in the Kubernetes cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.

PVs are resources in the cluster, just like pods, and they exist independently of any Pod that uses the PV.

They are meant to be used by one or more pods as persistent storage.

The administrator defines PVs with specifications including storage capacity, access modes, and other parameters such as the volume type, like NFS, AWS EBS, or Azure Disk.

***Persistent Volume Claim (PVC):***

A Persistent Volume Claim is a request for storage by a user or a pod within the cluster.

PVCs are how pods request specific storage resources from the cluster. Instead of pods directly accessing PVs, they request storage via PVCs.

When a pod needs access to persistent storage, it requests a PVC with specific requirements like storage capacity and access modes.

Once a PVC is created, Kubernetes finds an appropriate PV that matches the PVC requirements, binds the PVC to the PV, and then mounts the PV to the pod that requested it.

PVCs abstract away the underlying storage implementation details from the pod, allowing developers to work with storage resources in a more declarative manner.

In summary, Persistent Volumes are the actual storage resources within the Kubernetes cluster, while Persistent Volume Claims are requests for storage made by pods. PVCs act as an intermediary between pods and PVs, allowing for more flexible and dynamic allocation of storage resources within the cluster.\

In simple terms, Think of Persistent Volumes (PVs) as storage units in a building, like lockers or closets, where you can store your belongings. These storage units are set up by the building manager (or administrator) and come with different sizes and features, like small, medium, or large lockers, with different security levels.

Now, imagine you're a tenant in the building and you need storage space for your stuff. You don't want to deal with finding an empty locker or figuring out which one suits your needs. Instead, you go to the building manager's office and fill out a request form specifying your requirements—like how much space you need and what level of security you want. This request form is your Persistent Volume Claim (PVC).

The building manager then looks at your request and finds a suitable locker (Persistent Volume) that matches your needs. Once they find a match, they assign that locker to you and give you the keys. You can now use that locker to store your belongings. This process of matching your request with an available locker is called binding.

***So, in simple terms:***

Persistent Volumes (PVs) are like storage lockers.

Persistent Volume Claims (PVCs) are like requests you make to the building manager for a locker that suits your needs.

Binding is when the building manager assigns a locker to you based on your request.

Reference link: <https://kubernetes.io/docs/concepts/storage/persistent-volumes/#claims-as-volumes>

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

**ENV Variables in Kubernetes:**

In Kubernetes, environment variables are key-value pairs that can be injected into a container's environment at runtime. They are used to configure applications, pass sensitive information securely, and enable dynamic behavior within pods.

There are several ways to define environment variables in Kubernetes:

1. **Container Environment variable:** You can define environment variables directly within the container specification of a pod's YAML configuration file. For example:

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mycontainer

image: myimage

env:

- name: ENV\_VARIABLE\_NAME

value: "value"

1. **ConfigMaps:** ConfigMaps are Kubernetes objects used to store configuration data in key-value pairs. You can inject ConfigMap data into a pod's environment as environment variables. There are 2 phases involved in configuring configurationmaps, 1st create configmap 2nd Inject into pod; We do have 2 different ways to bring the configmap, 1st we can bring in “***Imperative way*** == without using a configmap definition file [here we can simply use the ***{kubectl create configmap}*** and specify the required arguments (ex: ***kubectl create configmap app-config –from-literal=APP\_COLOR=blue***”) this is not the recommended way as we have too many configuration items” and 2nd by “***Declarative way*** == by using a configmap definition file”

***Steps:***

1. Create a file (vi config-map.yaml)
2. Run the file (kubectl create –f config-map.yaml)
3. To view configmaps, run (kubectl get configmaps)
4. And to more, run (kubectl describe configmaps)

apiVersion: v1

kind: ConfigMap

metadata:

name: myconfigmap

data:

ENV\_VARIABLE\_NAME: value

Then, reference the ConfigMap in a pod specification:

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mycontainer

image: myimage

envFrom:

- configMapRef:

name: myconfigmap

1. **Secrets:** Secrets are similar to ConfigMaps but are designed to store sensitive information such as passwords, API tokens, and keys. There are 2 phases involved in configuring secrets, 1st create secrets 2nd Inject into pod; We do have 2 different ways to bring the secrets, 1st we can bring in “***Imperative way*** == without using a definition file [here we can simply specify the key value pairs ***{kubectl create secret generic}*** and specify the required arguments (ex: ***kubectl create secret generic \ app-secret –from-literal=DB\_Host=mysql***) however this could get complicated, when we have too many secrets to pass in” and 2nd by “***Declarative way*** == by using a secret definition file”

***Steps:***

1. Create a file (vi secret-data.yaml)
2. Run the file (kubectl create –f secret-data.yaml)
3. To view configmaps, run (kubectl get secrets)
4. And to more, run (kubectl describe secrets)
5. To know the values as well, run (kubectl get secret mysecret –o yaml)

apiVersion: v1

kind: Secret

metadata:

name: mysecret

data:

ENV\_VARIABLE\_NAME: <base64\_encoded\_value>

(ex:

DB\_Host: mysql

DB\_User: root

DB\_Password: passwd)

(DB\_Host: bXlzcWw=

DB\_User: cm9vdA==

DB\_Password: cGFzc3dk)

[as we can see here also we mentioned plain text, but in real-time we are going to pass the encoded value; in order to get the encoded value; go to the terminal, give command (echo –n ‘mysql’ | base64) and you’ll be provided with the encoded value that is what we are going to use here; and to decode the value, come to terminal and run the command (echo –n ‘bXlzcWw’ | base64 --decode)

Then, reference the secret in a pod specification:

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mycontainer

image: myimage

envFrom:

- secretRef:

name: mysecret

Overall, Environment variables provide a flexible way to configure applications running in Kubernetes pods, manage sensitive information securely, and enable communication between different components of a distributed system.

***Note on Secrets:***

1. Secrets are not encrypted, only encoded. [as it is easy for anyone to follow the decode method to get the secret]
2. Remember do not check-in secret object to SCM along with code.
3. Secrets are not encrypted in ETCD, [at this time we need to consider enabling Encryption at Rest]

Reference link: <https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/>

1. Anyone able to create pods/deployments in the same namespace can access the secrets [for this we need to consider configuring Role-based access control (RBAC)]
2. Consider 3rd party secrets store providers, such AWS, Azure, GCP, and Vault for more protection (not in Etcd, but in external secret provider)

**Istio in Kubernetes:**

Istio is an open-source service mesh platform that is used to connect, secure, control, and observe services in kubernetes and other platforms. It provides a uniform way to connect, manage, and secure Microservices. By using Istio we can visualize a running cluster; as we know Microservices architectures are complicated, Istio will give you a graphical view of your system right down to the Pod level, so it’s easy for us to see how it hangs together, where all the traffic’s going, where there are problems.

Service Mesh is not going to replace kubernetes, but this software we are going to use along with Kubernetes as an extra layer, Service Mesh is not related to Kubernetes, any kind of distributed architecture that is where we have multiple software components networking with each other, would benefit from service mesh. We are using it in Kubernetes because this tool considered to be more successful one for building distributed architectures, or in other words for Orchestration frameworks.

So let’s consider, we have out kubernetes setup, and there are many pods along with containers, and all this containers are networked together via service discovery mechanism to make network calls from any container to any other container. The problem here is, even though the pods handling and managing are good with kubernetes, the bad part here is visibility is not so clear when comes to all these interconnections between the pods. Consider for example, if some of the network requests are not satisfied here and if we are having 1000s of Microservices, we are going to have uncountable number of combinations of possible network calls between these pods. With standard kubernetes we don’t have any kind of visibility or control over the connections between the containers. So this is where a service mesh can help. (All of our network traffic running in our cluster are going to be routed through the service mesh software).

**Security (Certificates):**

**TLS Certificate:**

A certificate is used to guarantee trust between 2 parties during the transaction. [Ex: when a user tries to access a web servers, TLS certificates ensures that the communication between the user and the server is encrypted]

***Scenario:*** Without secure connectivity if a user access his online banking application, the credentials he types in would be sent in a plain text format, the hacker sniffing network traffic could easily retrieve the credentials and use it to hack into the user’s bank account; so it’s always important to encrypt the data using encryption keys. The data is using an encryption keys which is basically a random numbers and alphabets, you add the random number to your data and you encrypt it into a format that cannot be recognized, this data sent to the server, and the hacker sniffing the network gets the data but can’t do anything with it. However, it’s the same with server, where it won’t be able to decrypt without the key so a copy of the key must also sent, so this copy can decrypt and read the message; Since the key also sent over the same network the attacker can sniff that as well and decrypt the data, This is known as Symmetric encryption.

Symmetric encryption == it is a secure way of encryption, but since it uses the same key to encrypt and decrypt the data, and since the key has to be exchanged between the sender and the receiver, there is a risk of a hacker gaining access to the key, this is where asymmetric encryption comes into picture.

Asymmetric encryption == Instead of suing a single key to encrypt and decrypt data, asymmetric encryption uses a pair of keys (private & public key); now consider public as (key) format and private as (lock) format, The trick here is if you encrypt the data or lock the data, you can only open it with the associated key; So your key must always be secure with you and not be shared with anyone else it’s (private), but the lock is public and maybe shared with others,

***Scenario:*** You have a server in your environment and you don’t want to user passwords as they’re too risky, so we decide to user key pairs, (comes with a public and private key pair); we can do this by running ***“ssh-keygen”*** command as it creates 2 files (id\_rsa (private key), id\_rsa.pub (public key or public lock))

Just like last example, here we are going to securely transfer the keys using asymmetric encryption, by creating keys on the server using ssh-keygen; and then we are going to refer the public log

In order to open the public and private key, we can use ssl command

***(openssl genrsa –out my-bank.key 1024)***

***(openssl rsa –in my-bank.key –pubout > mybank.pem)***

When the user 1st access the web server using the HTTPS, he gets the public key from the server. Since, the hacker is sniffing all traffic let us assume he too gets a copy of the public key, The user’s browser then encrypts the symmetric key using the public key provided by the server, as we can see the symmetric key is not secure, the user send this to the server, and hacker also receives a copy, the server uses the private key to decrypt the message and retrieve the symmetric key from it, however, the hacker doesn’t have the private key to decrypt and retrieve the symmetric key from the message it received; the hacker got only the public key with which he can only lock or encrypt the message and not decrypt the message. The symmetric key is now only available to user and the server.

The hacker now looks for some ways to get access to our account, the only way to do that is by creating a form just like our application and type the data, and he hosts the application on his own server, to give an assurance the hacker generates his own set of public and private key pair and configures them on his web server. Somehow the hacker tweak out network to reroute our traffic to back goes to his server, when we open the browser and type the URL that’s going to give us with the same output login page, here without knowing this we are going to provide the username and password; during these kinds of situation in order to verify the key we receive is a legitimate or not (we need to look closely because here we are not going to receive the key instead we are going to receive the certificate that has the key in it), every certificate has a name on it, the person or subject to whom the certificate is issued to that is very important as this is filed to validate their identity.

But anyone can generate a certificate like this, so how can we look and verify it as legit; (we need to verify the signed and issued authority of the certificate) if an individual creating a certificate they need to sign it by themselves, that is known as a self-signed certificate; anyone looking at this certificate can confirm that it is not a safe certificate; where most of the time the browser itself do this job for us as it will check with the server and validates it; means on the top URL itself that will mention (Not secure)

How a legitimate certificates are created by companies where it was trusted by the web browsers, How to get a certificate that was signed by someone with authority? That’s where certificate authorities or CA’s comes in; there are known organizations that can sign and validate certificates for you, some of the popular one are Symantec, DigiCert, Comodo etc.,

How this works?

We generate a certificate signing request (CSR) using the key we generated earlier along with the domain name of our website, the certificate authorities are going to verify our details and if everything fine then they sign the certificate and sent it back; we can use this certificate signed by CA and so we don’t have any issues with the browsers.

What if the CA itself fake? Consider our certificate signed by Symantec, How would the browser know that Symantec actually signed or someone who’s says Symantec have signed? AS we know the CA’s themselves has a set of public and private key pairs, the CA’s use their private keys to sign the certificates, the public keys of all the CAs are built in to the browsers, through that browser i9s going to validate that the certificate is legitimate or not.

TLS (Transport Layer Security) certificates are used to encrypt communication between different components of the Kubernetes cluster, ensuring that data exchanged between them is secure and protected from eavesdropping or tampering. TLS certificates play a crucial role in securing various aspects of Kubernetes, including communication between API server and clients, inter-node communication, and communication between Kubernetes components.

Some common use cases of TLS certificates in Kubernetes are:

1. API server
2. Inter-Node communication
3. Service Communication
4. Ingress
5. Authentication and Authorization

SSL (Secure Sockets Layer) is a predecessor of TLS (Transport Layer Security), and the term is often used interchangeably with TLS. In Kubernetes, SSL/TLS plays a crucial role in securing communication between various components within the cluster, as well as between the cluster and external entities. Here's what SSL/TLS does in Kubernetes.