**K8s interview questions**

**What is autoscaling In k8s?**

Ans:- autoscaling in Kubernetes (often abbreviated as K8s) refers to the ability of the Kubernetes cluster to automatically adjust the number of running instances (pods) based on observed or defined metrics. The goal of autoscaling is to ensure that the application running on the cluster can handle varying levels of load efficiently without manual intervention.

There are two main types of autoscaling in Kubernetes:

1. \*\*Horizontal Pod Autoscaler (HPA):\*\*

- \*\*Objective:\*\* Scales the number of pod replicas in a deployment or replica set.

- \*\*How it works:\*\* Monitors specified metrics, such as CPU utilization or custom metrics, and adjusts the number of pod replicas to maintain the desired metric values.

- \*\*Configuration:\*\* You define the scaling behavior by setting up an HPA resource in Kubernetes, specifying the target metric and the desired metric values.

Example HPA YAML configuration:

```yaml

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: example-hpa

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: example-deployment

minReplicas: 2

maxReplicas: 10

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 80

```

2. \*\*Vertical Pod Autoscaler (VPA):\*\*

- \*\*Objective:\*\* Adjusts the resource requests and limits of individual pod containers based on observed usage.

- \*\*How it works:\*\* Monitors resource utilization of individual pods and adjusts the resource requests and limits to optimize resource allocation.

- \*\*Configuration:\*\* You define the scaling behavior by setting up a VPA resource in Kubernetes. It can be used to automatically adjust CPU and memory requests and limits.

Example VPA YAML configuration:

```yaml

apiVersion: autoscaling.k8s.io/v1

kind: VerticalPodAutoscaler

metadata:

name: example-vpa

spec:

targetRef:

apiVersion: "apps/v1"

kind: "Deployment"

name: "example-deployment"

updatePolicy:

updateMode: "Auto"

```

Autoscaling is a crucial feature in Kubernetes as it allows clusters to efficiently allocate resources based on demand, ensuring optimal performance and resource utilization. It simplifies the management of applications by automatically adjusting the number of instances in response to changes in load, providing a more dynamic and adaptive infrastructure.

2) what is k8s statefull set and why would you use one?

Ans:-

A StatefulSet in Kubernetes is a workload API object that is designed to manage stateful applications. It is an extension of a Deployment and provides guarantees about the ordering and uniqueness of pods. StatefulSets are often used for applications that require stable network identities, stable storage, and ordered deployment and scaling.

Here are some key characteristics and reasons why you might use a StatefulSet in Kubernetes:

1. \*\*Stable Network Identities:\*\*

- Each pod in a StatefulSet gets a unique and stable hostname based on the name of the StatefulSet and a unique index. For example, if you have a StatefulSet named "web" and you scale it to three replicas, the pods will be named "web-0," "web-1," and "web-2." This provides stable network identities, which can be crucial for certain applications that rely on consistent naming.

2. \*\*Stable Storage:\*\*

- StatefulSets support the use of persistent volumes (PVs) and persistent volume claims (PVCs). Each pod in a StatefulSet gets its own PVC, and the data stored in the PV is associated with the pod's unique index. This ensures that even if a pod is rescheduled or moved to a different node, it can still access its persistent storage.

3. \*\*Ordered Deployment and Scaling:\*\*

- StatefulSets deploy pods in a predictable and ordered manner. Each pod is created one at a time, and the next pod is only started once the previous one is running and ready. This ordering can be important for applications that have dependencies or require a specific startup sequence.

4. \*\*Scaling:\*\*

- When scaling a StatefulSet, the pods are scaled up or down in a way that maintains the ordering and uniqueness of their identities. This is different from Deployments, where pods can be scaled independently without specific identities.

Here's an example of a simple StatefulSet YAML configuration:

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

serviceName: "nginx"

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

ports:

- containerPort: 80

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: ["ReadWriteOnce"]

resources:

requests:

storage: 1Gi

…

In summary, you would use a StatefulSet in Kubernetes for stateful applications that require stable network identities, stable storage, and ordered deployment and scaling. Examples of such applications include databases, messaging systems, and other stateful workloads. StatefulSets provide a higher level of abstraction and management for such applications compared to regular Deployments.

3) what is k8s loadbalancer and how does it works?

Ans:-

In Kubernetes (K8s), a LoadBalancer is a service type that exposes a set of pods to the internet or a specified network by automatically provisioning an external load balancer. The primary purpose of a LoadBalancer service is to distribute incoming network traffic across multiple backend pods to ensure high availability and reliability.

Here's how a LoadBalancer works in Kubernetes:

1. \*\*Service Definition:\*\*

- You define a Kubernetes Service of type `LoadBalancer` in your cluster by creating a Service YAML file. This file specifies the type of service, the selector for identifying the backend pods, and the port configuration.

Example LoadBalancer service definition:

```yaml

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: my-app

ports:

- protocol: TCP

port: 80

targetPort: 8080

type: LoadBalancer

```

2. \*\*Service Creation:\*\*

- When you create the Service in the Kubernetes cluster, the cloud provider's integration (like AWS, GCP, or Azure) automatically provisions an external load balancer. The load balancer is associated with a public IP address, DNS name, or other network-specific endpoint.

3. \*\*Backend Pod Discovery:\*\*

- The LoadBalancer service monitors the set of pods that match the specified selector (`app: my-app` in the example). These pods constitute the backend for the service.

4. \*\*Traffic Distribution:\*\*

- The external load balancer forwards incoming traffic to one of the available backend pods. The load balancer performs load distribution based on a chosen algorithm (such as round-robin or least connections) to balance the load evenly across the backend pods.

5. \*\*Health Checks:\*\*

- The LoadBalancer service often performs health checks on the backend pods to ensure that only healthy pods receive traffic. If a pod becomes unhealthy, the load balancer stops directing traffic to that pod until it becomes healthy again.

6. \*\*Network Address Translation (NAT):\*\*

- The load balancer typically performs Network Address Translation (NAT) to hide the internal IP addresses of the backend pods. Clients interact with the load balancer's external IP address or DNS name, and the load balancer forwards traffic to the appropriate backend pod.

7. \*\*Scaling:\*\*

- As you scale your application by adding or removing pods, the LoadBalancer service dynamically adjusts the pool of backend pods. This ensures that traffic is distributed evenly across the available instances.

8. \*\*Cleanup:\*\*

- When you delete the LoadBalancer service, the cloud provider's integration deallocates the external load balancer, releasing any associated resources.

It's important to note that the details of how a LoadBalancer is provisioned and configured depend on the cloud provider. Different cloud providers have their own LoadBalancer implementations and configurations. In the example above, the external IP address and DNS name would be specific to the cloud provider's load balancer service.

4) what is k8s ingress and how it works?

Ans:-

In Kubernetes, an Ingress is an API object that provides HTTP and HTTPS routing to services based on rules. It acts as a layer 7 (application layer) load balancer, allowing you to define how external HTTP/S traffic should be directed to different services within your cluster. Ingress provides a way to expose services to the outside world and manage routing and URL paths.

Here's how Ingress works in Kubernetes:

1. \*\*Ingress Resource Definition:\*\*

- You define an Ingress resource by creating an Ingress YAML file. This file specifies rules for routing traffic, including hostnames, paths, and the corresponding backend services.

Example Ingress resource definition:

```yaml

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: my-ingress

spec:

rules:

- host: myapp.example.com

http:

paths:

- path: /app

pathType: Prefix

backend:

service:

name: myapp-service

port:

number: 80

```

In this example, traffic to `myapp.example.com/app` will be directed to the `myapp-service` on port 80.

2. \*\*Ingress Controller:\*\*

- An Ingress Controller is a component in the Kubernetes cluster responsible for implementing the rules defined in the Ingress resource. The Ingress Controller watches for changes to Ingress resources and configures the underlying load balancer or other routing mechanisms accordingly.

3. \*\*Routing and Load Balancing:\*\*

- When a request comes to the specified hostname and path defined in the Ingress resource, the Ingress Controller routes the traffic to the appropriate backend service. This may involve load balancing the traffic across multiple instances of the backend service.

4. \*\*TLS Termination (Optional):\*\*

- Ingress can also handle TLS termination, allowing you to secure your applications using HTTPS. You can specify TLS certificates in the Ingress resource to enable encrypted communication between clients and the Ingress Controller.

Example TLS configuration in Ingress:

```yaml

spec:

tls:

- hosts:

- myapp.example.com

secretName: myapp-tls-secret

```

This example specifies that TLS certificates stored in the `myapp-tls-secret` Secret should be used for securing communication with `myapp.example.com`.

5. \*\*Path-Based Routing:\*\*

- Ingress supports path-based routing, allowing you to route traffic to different backend services based on URL paths. This is demonstrated in the example Ingress resource definition, where requests to `/app` are routed to the `myapp-service`.

6. \*\*Wildcard Hosts and Default Backends:\*\*

- Ingress allows you to use wildcard hosts (`\*`) and define default backends. Wildcard hosts enable routing for any subdomain, and the default backend handles requests that do not match any defined rules.

Example wildcard host and default backend:

```yaml

spec:

rules:

- host: "\*"

http:

paths:

- path: "/"

pathType: Prefix

backend:

service:

name: default-backend

port:

number: 80

```

In this example, any request not matching other rules will be directed to the `default-backend` service.

In summary, Ingress in Kubernetes provides a flexible and high-level way to manage external access to services within the cluster. It allows you to define routing rules, handle TLS termination, and manage traffic to different backend services based on URL paths and hostnames. The Ingress Controller is a crucial component that interprets and enforces these rules in the cluster.

5) what is k8s container and how is differ from k8s pod?

Ans:- Kubernetes (often abbreviated as K8s) is an open-source container orchestration platform used for automating the deployment, scaling, and management of containerized applications. Containers are lightweight, portable, and self-sufficient units that encapsulate an application and its dependencies.

In the context of Kubernetes, a "pod" is the smallest and simplest unit in the Kubernetes object model. It represents a single instance of a running process in a cluster and encapsulates one or more containers. Containers within a pod share the same network namespace, allowing them to communicate with each other using localhost. They can also share storage volumes, making it easier for them to exchange data.

So, to clarify:

1. \*\*Container\*\*: A container is a lightweight, standalone, and executable software package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools.

2. \*\*Pod\*\*: A pod is the smallest deployable unit in Kubernetes and represents a single instance of a running process in a cluster. It encapsulates one or more containers that are tightly coupled and share the same resources and network namespace.

In summary, a pod is a higher-level abstraction that encapsulates one or more containers, and these containers share the same network and storage context. Containers, on the other hand, are the executable packages that contain the application and its dependencies. In a Kubernetes cluster, you deploy and manage pods rather than individual containers to ensure the coordinated operation of containerized applications.

6) what is k8s operator and how it works?

**Ans:-** In Kubernetes, an "operator" refers to a method of packaging, deploying, and managing a Kubernetes application. It extends the Kubernetes API to create, configure, and manage instances of complex stateful applications. Essentially, operators are software extensions to Kubernetes that leverage custom resources to automate the deployment and management of applications.

Here's how operators work:

1. \*\*Custom Resources (CRs)\*\*: Operators make use of Custom Resources, which are extensions of the Kubernetes API. These custom resources define new object types specific to your application. For example, if you're managing a database with an operator, you might create a custom resource for a database instance.

2. \*\*Custom Controllers\*\*: Operators include custom controllers that watch for changes to these custom resources. When a custom resource is created, modified, or deleted, the controller takes action.

3. \*\*Reconciliation Loop\*\*: The core concept of an operator is the reconciliation loop. The operator's controller runs a continuous loop to ensure that the current state of the system matches the desired state specified in the custom resources. If there are any discrepancies, the operator takes corrective actions to bring the system back to the desired state.

4. \*\*Automation and Lifecycle Management\*\*: Operators automate various tasks such as deployment, scaling, updates, and failure recovery. They encapsulate operational knowledge about an application, allowing for complex, stateful applications to be managed more effectively within Kubernetes.

5. \*\*Example Use Cases\*\*:

- \*\*Database Management\*\*: Operators can automate database provisioning, scaling, backup, and recovery.

- \*\*Application Deployment\*\*: Operators can manage the lifecycle of custom applications, handling tasks like rolling updates and configuration changes.

- \*\*Monitoring and Logging\*\*: Operators can automate the deployment and configuration of monitoring and logging solutions.

Here's a simplified example of an operator's workflow:

- You define a custom resource, such as a "Database," in a YAML file.

- You apply this YAML file to your Kubernetes cluster.

- The operator's controller detects the custom resource and triggers the reconciliation loop.

- The operator takes actions to ensure that the database is deployed, scaled, and configured as specified in the custom resource.

- The operator continues to monitor and reconcile the state, making adjustments as needed.

Operators enhance the manageability of complex applications in Kubernetes by encoding operational knowledge into the system. They bring a higher level of automation and consistency to the deployment and management of applications, reducing manual intervention and potential errors.

7) what is k8s helm chart and how it works?

Ans:- Kubernetes (often abbreviated as K8s) is an open-source container orchestration platform for automating the deployment, scaling, and management of containerized applications. Helm, on the other hand, is a package manager for Kubernetes applications. Helm uses a packaging format called charts, which are a collection of pre-configured Kubernetes resources.

Here's a breakdown of what a Helm chart is and how it works:

1. \*\*Helm Chart:\*\*

- A Helm chart is a collection of pre-configured Kubernetes resources, such as deployments, services, and config maps, bundled together.

- Charts are packaged as compressed archives (typically `.tgz` files) and contain a combination of YAML files and template files.

- The YAML files define the Kubernetes resources, and the template files use the Go templating language to parameterize and customize the resource configurations.

2. \*\*Structure of a Helm Chart:\*\*

- A typical Helm chart directory has a predefined structure. Some of the important files and directories include:

- `Chart.yaml`: Metadata about the chart, such as its name, version, and description.

- `values.yaml`: Default configuration values for the chart. These can be overridden by users during installation.

- `templates/`: Directory containing the template files for Kubernetes resource definitions.

- `charts/`: Directory where dependencies (other charts) can be stored.

3. \*\*Helm Workflow:\*\*

- \*\*Chart Creation:\*\* Developers create Helm charts to package their Kubernetes applications. They define the necessary Kubernetes resources, template them as needed, and package them into a chart.

- \*\*Chart Distribution:\*\* Charts can be distributed via various methods, such as Helm repositories, which are collections of charts hosted on a server. Developers or organizations can publish their charts to a repository.

- \*\*Installation:\*\* Users deploy applications on Kubernetes using Helm by installing charts. During installation, users can customize configurations using the values provided in the `values.yaml` file or by passing values directly via the command line.

- \*\*Template Rendering:\*\* When a Helm chart is installed, Helm renders the templates using the specified values, generating Kubernetes manifest files with the customized configurations.

- \*\*Resource Deployment:\*\* Helm then uses Kubernetes API to deploy the rendered resources to the cluster.

4. \*\*Benefits of Helm:\*\*

- \*\*Reusability:\*\* Charts can be shared and reused across different projects and organizations.

- \*\*Versioning:\*\* Charts can have versions, making it easier to manage and track changes over time.

- \*\*Configuration Management:\*\* Helm allows users to parameterize and customize applications' configurations during installation.

In summary, Helm provides a convenient way to package, distribute, and deploy Kubernetes applications with predefined configurations, making it easier to manage complex applications and promote reusability.

8) how does k8s handles container networking?

Ans:- Kubernetes (K8s) manages container networking by providing a network overlay that allows containers running on different nodes in a cluster to communicate with each other seamlessly. Kubernetes uses the Container Network Interface (CNI) to achieve this, which is a standard interface between container runtimes and network plugins.

Here's an overview of how Kubernetes handles container networking:

1. \*\*Pods:\*\*

- The basic unit of deployment in Kubernetes is a Pod, which is the smallest deployable unit and can contain one or more containers.

- Containers within a Pod share the same network namespace, allowing them to communicate with each other using `localhost` and share the same IP address.

2. \*\*Cluster Networking:\*\*

- Pods in a Kubernetes cluster need to communicate with each other, regardless of the node they are running on.

- Kubernetes assigns each Pod a unique IP address from the cluster-wide address range.

- Nodes in the cluster are responsible for routing traffic between Pods, and this routing is typically handled by a container network plugin.

3. \*\*Container Network Interface (CNI):\*\*

- CNI is a standardized interface for networking plugins in container runtimes. Kubernetes uses CNI to integrate with various network plugins.

- CNI plugins are responsible for configuring the network interfaces of containers within a Pod, setting up routes, and managing IP address assignments.

4. \*\*Network Plugins:\*\*

- Kubernetes supports various network plugins that implement the CNI specification. Examples include Calico, Flannel, Weave, and others.

- These plugins provide different ways to handle networking, such as overlay networks, bridge networks, and direct routing.

5. \*\*Overlay Networks:\*\*

- Many Kubernetes deployments use overlay networks to enable communication between Pods on different nodes.

- Overlay networks encapsulate the communication between containers in additional headers, allowing them to be transmitted across nodes without the need for the underlying network infrastructure to be aware of the Pod-level communication.

6. \*\*Service Abstraction:\*\*

- Kubernetes Services provide an abstraction for exposing a set of Pods as a network service. Services have stable IP addresses and DNS names that allow other Pods within the cluster to access them.

7. \*\*Ingress Controllers:\*\*

- Ingress controllers provide HTTP and HTTPS routing to Services in a Kubernetes cluster. They act as an entry point for external traffic, managing rules for routing requests to the appropriate Services.

In summary, Kubernetes manages container networking through the use of CNI plugins that configure network interfaces for Pods, assign unique IP addresses, and enable communication between Pods across different nodes. Network plugins, such as overlay networks, play a crucial role in ensuring seamless connectivity within the cluster. The use of CNI provides flexibility, allowing Kubernetes to integrate with various networking solutions based on the specific requirements of the deployment.

9) how can you scale k8s application horizontally?

Ans:- Scaling a Kubernetes application horizontally involves increasing the number of instances (pods) of your application to distribute the load and improve performance. Kubernetes provides mechanisms to easily scale applications horizontally. Here are the key steps:

### 1. \*\*Replica Sets:\*\*

- Kubernetes uses Replica Sets to manage the desired number of replicas (pods) for a given application. You define a Replica Set with a specified number of replicas, and Kubernetes ensures that this number of replicas is always running.

### 2. \*\*Deployment Scaling:\*\*

- Deployments are a higher-level abstraction that manages Replica Sets. You can use Deployments to scale your application by updating the replica count. Here's an example:

```bash

kubectl scale deployment <deployment-name> --replicas=<desired-replica-count>

```

This command will update the deployment's replica count, and Kubernetes will automatically adjust the number of pods to match the desired count.

### 3. \*\*Autoscaling:\*\*

- Kubernetes supports Horizontal Pod Autoscaling (HPA), which automatically adjusts the number of replicas in a Deployment or Replica Set based on observed metrics like CPU utilization or custom metrics.

```yaml

apiVersion: autoscaling/v2beta2

kind: HorizontalPodAutoscaler

metadata:

name: <hpa-name>

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: <deployment-name>

minReplicas: <min-replicas>

maxReplicas: <max-replicas>

metrics:

- type: Resource

resource:

name: cpu

targetAverageUtilization: 50

```

In this example, the HPA is configured to adjust the replica count to maintain an average CPU utilization of 50%.

### 4. \*\*Manual Scaling:\*\*

- You can manually update the replica count in the deployment manifest and apply the changes using `kubectl apply`:

```yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: <deployment-name>

spec:

replicas: <desired-replica-count>

# other deployment configurations

```

Then apply the changes:

```bash

kubectl apply -f deployment.yaml

```

### 5. \*\*Load Balancing:\*\*

- Ensure that your service is set up with a type that supports load balancing (e.g., `LoadBalancer` or `NodePort`). This helps distribute incoming traffic across all running replicas.

### 6. \*\*Monitoring:\*\*

- Implement monitoring and alerting to track the performance of your application and pods. This helps in making informed decisions about scaling based on metrics like CPU usage, memory usage, and custom application metrics.

By combining these techniques, you can effectively scale your Kubernetes applications horizontally, ensuring that the desired number of replicas is running to handle the load and traffic efficiently. Additionally, autoscaling provides the benefit of automatic adjustments based on observed metrics, making it a dynamic and responsive scaling solution.

10) how can you secure k8s cluster?

Ans:- Securing a Kubernetes (K8s) cluster is crucial to protect against potential vulnerabilities and unauthorized access. Here are some key practices and strategies for securing a Kubernetes cluster:

1. \*\*Role-Based Access Control (RBAC):\*\*

- Implement RBAC to control access to the Kubernetes API and resources. Define roles and role bindings to ensure that users and service accounts have the minimum necessary permissions.

2. \*\*Pod Security Policies (PSP):\*\*

- Use Pod Security Policies to restrict the capabilities and permissions of pods within the cluster. Define policies that control aspects like privilege escalation, volume mounts, and host namespaces.

3. \*\*Network Policies:\*\*

- Leverage Network Policies to control the communication between pods. Define rules to allow or deny traffic based on specific criteria such as pod labels, namespaces, or IP ranges.

4. \*\*Secure API Server:\*\*

- Use TLS to encrypt communication between components in the Kubernetes cluster, especially the API server. Ensure that only secure protocols are used.

5. \*\*Etcd Encryption:\*\*

- If possible, enable encryption for data at rest in etcd, the distributed key-value store used by Kubernetes to store cluster data. This helps protect sensitive information stored in etcd.

6. \*\*Container Image Security:\*\*

- Use only trusted container images from reputable sources. Regularly scan images for vulnerabilities, and keep them up-to-date. Implement image signing and verification when possible.

7. \*\*Secrets Management:\*\*

- Properly manage and secure Kubernetes secrets. Avoid storing sensitive information directly in YAML files. Use tools like Kubernetes Secrets or external secret management systems.

8. \*\*Pod Security Context:\*\*

- Utilize Pod Security Context to set security-related attributes for pods, such as user and group IDs, file permissions, and SELinux options.

9. \*\*Limit Privileges:\*\*

- Run containers with the least privilege necessary. Avoid running containers as the root user, and use security contexts to set appropriate user IDs.

10. \*\*Audit Logging:\*\*

- Enable audit logging for the Kubernetes API server to track user activity and detect potential security incidents. Centralize and monitor the audit logs.

11. \*\*Network Segmentation:\*\*

- Segment the network to isolate different parts of the cluster. Use network policies to restrict communication between namespaces and clusters.

12. \*\*Update and Patch:\*\*

- Regularly update and patch both the Kubernetes cluster components and the underlying operating system. Stay informed about security updates and apply them promptly.

13. \*\*Harden Nodes:\*\*

- Secure the underlying nodes by disabling unnecessary services, applying operating system-level security configurations, and using tools like kube-bench to check for security best practices.

14. \*\*API Server Authentication:\*\*

- Use strong authentication mechanisms for the API server, such as client certificates, bearer tokens, or integration with external authentication providers like LDAP or OIDC.

15. \*\*Limit External Access:\*\*

- Limit external access to the Kubernetes API server and control access through firewalls and network policies. Consider using VPNs or private network connections.

Implementing a combination of these security practices helps create a robust defense-in-depth strategy for securing a Kubernetes cluster. Regularly audit and update security measures to stay ahead of potential threats and vulnerabilities.