Kubernetes Interview Questions

Question 1:

How to run Kubernetes locally?

Answer:

You can run Kubernetes locally using tools like Minikube, Kind (Kubernetes in Docker), or Docker Desktop with Kubernetes enabled. These tools help create a single-node Kubernetes cluster on your local machine for development and testing purposes.

Question 2:

What is Kubernetes Load Balancing?

Answer:

Kubernetes load balancing ensures even distribution of incoming traffic across multiple pods or containers within a Kubernetes cluster. This is achieved through services like NodePort, ClusterIP, LoadBalancer, or Ingress, which help route traffic to the appropriate pods based on defined rules and configurations.

Question3:

What the following in the Deployment configuration file mean?

```
spec:
  containers:
    - name: USER_PASSWORD
      valueFrom:
        secretKeyRef:
          name: some-secret
          key: password
```

Answer:

In the Deployment configuration file, the section you provided specifies the environment variable `USER_PASSWORD` for a container within a pod. The value of this

environment variable is retrieved from a Kubernetes Secret named `some-secret`, and specifically from the key named `password` within that secret. This allows sensitive information, like passwords or API keys, to be securely stored and injected into containers at runtime.

Question 4:

How to troubleshoot if the POD is not getting scheduled?

Answer:

In K8's scheduler is responsible to spawn pods into nodes. There are many factors that can lead to unstartable POD. The most common one is running out of resources, use the commands like kubectl describe <POD> -n <Namespace> to see the reason why POD is not started. Also, keep an eye on kubectl to get events to see all events coming from the cluster.

Question 5:

How to run a POD on a particular node?

Answer:

Various methods are available to achieve it.

- **nodeName:** specify the name of a node in POD spec configuration, it will try to run the POD on a specific node.
- **nodeSelector:** Assign a specific label to the node which has special resources and use the same label in POD spec so that POD will run only on that node.
- **nodeaffinities:** required DuringSchedulingIgnoredDuringExecution, preferredDuringSchedulingIgnoredDuringExecution are hard and soft requirements for running the POD on specific nodes. This will be replacing nodeSelector in the future. It depends on the node labels.

Question 6:

How to turn the service defined below in the spec into an external one?

spec:

```
  selector:
    app: some-app
  ports:
    - protocol: UDP
      port: 8080
      targetPort: 8080
```

Answer:

Adding type: LoadBalancer and nodePort as follows:

```
spec:
 selector:
   app: some-app
 type: LoadBalancer
 ports:
   - protocol: UDP
     port: 8080
     targetPort: 8080
     nodePort: 32412
```

Question 7:

How do you test a manifest without actually executing it?

Answer:

use --dry-run flag to test the manifest `kubectl apply -f my-manifest.yaml --dry-run=client`

Question 8:

How do you initiate a rollback for an application?

Answer:

- Kubernetes and kubectl offer a simple mechanism to roll back changes to resources like Deployments, StatefulSets, and DaemonSets[3].
- To initiate a rollback using kubectl, you can use the `kubectl rollout undo deploy <deploymentname>` command[1]

Question 9:

How do you package Kubernetes applications?

Answer:

To package Kubernetes applications, you can use Helm Charts, which simplify the process of packaging and deploying common applications on Kubernetes. Helm is a package manager for Kubernetes that allows users to templatize their Kubernetes manifests and provide configuration parameters.

Question 10:

What is node affinity and pod affinity?

Answer:

Node affinity in Kubernetes is for scheduling pods based on node properties, while pod affinity is for scheduling pods based on the presence of other pods.

Question 11:

How do you drain the traffic from a Pod during maintenance?

Answer:

To drain traffic from a Kubernetes pod during maintenance:

1. Use `kubectl drain` to mark the node as unschedulable and evict pods
2. Leverage Kubernetes maintenance mode to stop new pods on the node
3. Configure readiness and liveness probes to manage traffic routing
4. Implement graceful shutdown with the `preStop` hook

Question 12:

I have one POD and inside 2 containers are running one is Nginx and another one is wordpress So, how can access these 2 containers from the Browser with IP address?

Answer:

To access the Nginx and Wordpress containers running in a single Kubernetes pod from the browser using the IP address:

- Use the pod's IP address with the respective container ports:Nginx:

Question13:

If I have multiple containers running inside a pod, and I want to wait for a specific container to start before starting another one.

Answer:

To wait for a specific container to start before starting another in a Kubernetes pod:

1. Use Init Containers to perform setup tasks before the main containers
2. Leverage the `kubectl wait` command with the `--for=condition=Ready` flag to wait for a container to be ready

Question 14:

What is the impact of upgrading the kubelet if we leave the pods on the worker node - will it break running pods? Why?

Answer:

Upgrading kubelet without draining the node will cause all pods on the node to stop and restart, potentially leading to temporary unavailability. This is because the kubelet stops querying the API during the upgrade, affecting container operations. It's recommended to drain the node before upgrading kubelet to gracefully migrate pods and avoid disruption.

Question 15:

How service that selects apps based on the label and has an externalIP?

Answer:

To create a Kubernetes service that selects apps based on labels and has an externalIP:

- Define a service with a selector that matches the labels of the target pods.
- Specify the externalIP field in the service spec to assign an external IP address.
- The service will route traffic to the pods with the matching labels.

Question 16:

Does the container restart When applying/updating the secret object (kubectl apply -f mysecret.yml)? If not, how is the new password applied to the database?

Answer:

No, the container does not automatically restart when applying or updating the secret object using `kubectl apply -f mysecret.yml`. To apply the new password to the database without container restart, you can use mechanisms like environment variable reloading or application-specific mechanisms to pick up the new secret values dynamically without requiring a container restart.

Question 17:

How should you connect an app pod with a database pod?

Answer:

To connect an app pod with a database pod in Kubernetes:

1. Create a Service for the database pod
2. Update the app configuration to use the database Service hostname and port
3. Verify the connectivity between the app and database

Question 18:

How to configure a default ImagePullSecret for any deployment?

Answer:

To configure a default ImagePullSecret for any deployment in Kubernetes:

1. Create an ImagePullSecret with container registry credentials
2. Patch the default service account to use the ImagePullSecret

Question 19:

If you have a pod that is using a ConfigMap which you updated, and you want the container to be updated with those changes, what should you do?

Answer:

To update a container with changes made to a ConfigMap it is using, you need to restart the pod. Simply updating the ConfigMap does not automatically update the container - you must trigger a pod restart for the changes to take effect.

Question 20:

How does Kubernetes handle resource management (CPU, memory)? How can resource requests and limits be used effectively?

Answer:

Kubernetes manages resource allocation (CPU, memory) using resource requests and limits. Requests ensure pods get the necessary resources, while limits prevent them from consuming too much. Effective usage involves setting requests to what pods need for normal operation and limits to prevent excessive consumption, ensuring efficient resource utilization

Question 21:

Explain how you would troubleshoot a pod that is failing to start or is experiencing resource issues.

Answer:

To troubleshoot a pod failing to start or facing resource issues in Kubernetes, follow these steps:

1. Check Pod Status: Use the `kubectl get pods` command to check the status of the pod. Look for any error messages or warnings indicating why the pod failed to start.

2. View Pod Logs: Retrieve the logs of the pod using `kubectl logs <pod_name>`. Analyze the logs for any error messages or clues about what might be causing the issue.

3. Inspect Resource Requests and Limits: Review the resource requests and limits set for the pod in its YAML configuration file. Ensure they are appropriate for the pod's requirements. Adjust them if necessary.

4. Node Resource Utilization: Check the resource utilization of the nodes in the Kubernetes cluster using tools like `kubectl top nodes`. Ensure that there are enough available resources (CPU, memory) on the nodes to schedule the pod.

5. Events: Use `kubectl describe pod <pod_name>` to view detailed information about the pod, including any events related to its scheduling or execution. Look for events indicating resource shortages or scheduling failures.

6. Pod Affinity/Anti-affinity: If the pod has pod affinity or anti-affinity rules defined, ensure they are not preventing the pod from being scheduled due to node constraints.

7. Taints and Tolerations: Check if the node where the pod is supposed to run has any taints that might be preventing the pod from being scheduled. Ensure that the pod's tolerations are correctly configured to tolerate those taints if necessary.

8. Pod Security Policies: Verify if any pod security policies are affecting the pod's ability to start or access resources.

9. Restart Pod: Sometimes, simply restarting the pod using `kubectl delete pod <pod_name>` followed by `kubectl apply -f <pod_yaml>` can resolve transient issues.

10. Cluster-wide Issues: If multiple pods are experiencing similar issues, investigate if there are any cluster-wide issues such as network problems or service disruptions.

Question 22:

Describe the process of rolling out a new application deployment in Kubernetes. How do you ensure minimal downtime?

Answer:

To roll out a new application deployment in Kubernetes with minimal downtime:

- Update Docker Image: Update the Docker image of the application with the new changes.
- Update Deployment Manifest: Modify the deployment manifest (YAML file) with the new image version.
- Apply Changes: Use kubectl apply -f <deployment.yaml> to apply the changes to the deployment.
- Rolling Update Strategy: Ensure the deployment uses a rolling update strategy, which gradually replaces old pods with new ones.
- Replica Sets: Kubernetes creates a new replica set for the updated deployment while scaling down the old replica set.
- Pod Lifecycle: Pods from the new replica set are gradually scheduled and started, while pods from the old replica set are gradually terminated.
- Health Checks: Implement readiness and liveness probes to ensure that new pods are only considered ready once they are healthy.
- Monitor Progress: Monitor the deployment progress using kubectl rollout status <deployment_name>.
- Rollback Plan: Have a rollback plan in place in case of any issues, using kubectl rollout undo <deployment_name> if necessary.
- Ensure High Availability: Ensure high availability by running multiple replicas of the application to handle traffic during the rollout process.

Question 23:

Explain the concept of Kubernetes Ingress and how it can be used to manage external traffic routing.

Answer:

Kubernetes Ingress is an API object used to manage external access to services within a Kubernetes cluster. It acts as a traffic manager, routing incoming requests from outside the cluster to the appropriate services and pods inside the cluster. Here's how it works and how it manages external traffic routing:

- External Traffic Management: Ingress allows you to define rules that specify how external traffic should be directed to different services within the cluster based on factors like hostnames, paths, or request types.
- HTTP and HTTPS Routing: Ingress supports HTTP and HTTPS routing, enabling you to handle web traffic securely using TLS/SSL certificates.
- Load Balancing: Ingress controllers typically integrate with cloud providers' load balancers or provide their own load balancing mechanisms to distribute incoming traffic across multiple pods or nodes running the targeted services.
- Virtual Hosts and Path-Based Routing: Ingress allows you to define virtual hosts (domains) and route traffic based on the hostname specified in the HTTP request. Additionally, it supports path-based routing, enabling you to direct traffic to different services based on the URL path.
- TLS Termination: Ingress controllers can terminate TLS/SSL connections, decrypting incoming HTTPS requests and forwarding them to backend services over plain HTTP within the cluster.
- Reverse Proxy: Ingress controllers often act as reverse proxies, performing tasks like request forwarding, load balancing, and serving static content.
- Ingress Controllers: Kubernetes does not provide a built-in implementation of Ingress. Instead, various third-party or cloud-provider-specific Ingress controllers are available, such as NGINX Ingress Controller, Traefik, or AWS ALB Ingress Controller. These controllers implement the Ingress specification and manage external traffic routing according to the defined rules.
- Customization and Configuration: Ingress resources are defined using YAML manifests, allowing for easy customization and configuration of routing rules, TLS settings, and other parameters.

Question 24:

Have you had experience with tools like Helm for managing Kubernetes deployments and configurations? How does it simplify the process?

Answer:

Yes, I'm familiar with Helm, a popular package manager for Kubernetes that simplifies the process of managing deployments and configurations. Here's how it simplifies the process:

- Templating: Helm allows you to use templates (using Go templating language) to define Kubernetes manifests, enabling parameterization and reuse of configuration across different environments or deployments. This reduces duplication and simplifies maintenance.
- Charts: Helm packages Kubernetes resources into reusable units called charts. Charts encapsulate all the Kubernetes YAML manifests, along with default configurations, dependencies, and metadata, making it easy to share, distribute, and install applications.
- Dependency Management: Helm supports dependency management, allowing you to specify dependencies between different charts. Helm automatically manages the installation, upgrading, and deletion of dependencies when deploying or updating charts, simplifying complex application deployments.
- Versioning: Helm tracks the versions of installed charts and provides rollback capabilities, enabling you to easily revert to a previous version if an update causes issues. This simplifies the process of managing application upgrades and ensures rollback safety.
- Repositories: Helm supports the use of repositories to store and distribute charts. You can use public or private repositories to share charts within your organization or with the community, making it easy to discover and reuse pre-configured applications and configurations.
- Release Management: Helm manages deployments as releases, allowing you to install, upgrade, rollback, and delete releases with simple commands (helm install, helm upgrade, helm rollback, helm delete). This provides a consistent and reproducible way to manage application lifecycle in Kubernetes.
- Hooks: Helm supports pre-install, post-install, pre-upgrade, post-upgrade, and uninstall hooks, enabling you to execute custom scripts or actions during different stages of the deployment lifecycle. This allows for additional customization and integration with external systems.

Helm Interview Questions

Question 1:

What is Helm and how does it work in the context of Kubernetes?

Answer:

Helm is a package manager for Kubernetes that simplifies the process of deploying, managing, and scaling applications on Kubernetes clusters. It streamlines the management of Kubernetes manifests, which are YAML files describing Kubernetes resources such as deployments, services, and ingress rules.

In the context of Kubernetes, Helm operates with the following key components:

- Charts: A Helm package is called a chart. A chart is a collection of files that describe a set of Kubernetes resources. It includes templates for Kubernetes manifests, default configurations, dependency information, and metadata. Charts enable you to package, version, and distribute applications and configurations in a reusable and shareable manner.
- Templates: Helm uses Go templating to generate Kubernetes manifests dynamically. Templates allow you to parameterize configurations and customize deployments for different environments or use cases. This makes it easier to manage configurations across multiple deployments and reduces duplication.
- Repositories: Helm repositories are collections of charts that you can browse, search, and install. Helm comes with a default stable repository containing a wide range of popular charts for common applications and services. You can also set up private repositories to store and share custom charts within your organization.
- Commands: Helm provides a command-line interface (CLI) for interacting with charts and managing deployments. Common Helm commands include helm install to install a chart, helm upgrade to upgrade a release, helm rollback to roll back to a previous version, and helm delete to uninstall a release. Helm CLI

simplifies the deployment lifecycle by abstracting away the complexity of interacting with Kubernetes APIs directly.

- Release Management: Helm manages deployments as releases, which are instances of installed charts. Each release has a unique name and version, allowing you to track and manage the lifecycle of applications deployed on Kubernetes clusters. Helm supports versioning, rollback, and deletion of releases, providing a robust mechanism for managing application deployments.

Question 2:

Explain the benefits of using Helm for managing Kubernetes applications in a production environment.

Answer:

Using Helm for managing Kubernetes applications in a production environment offers several benefits:

- Standardization: Helm provides a standardized way to package, deploy, and manage Kubernetes applications. By defining applications as Helm charts, teams can establish consistent deployment practices across their organization, reducing errors and improving reliability.
- Reusability: Helm charts encapsulate Kubernetes manifests, default configurations, and dependencies into reusable units. This promotes code reuse and modularity, allowing teams to easily share and distribute applications within their organization or with the wider community. Reusing Helm charts saves time and effort by eliminating the need to recreate configurations for each deployment.
- Versioning and Rollback: Helm tracks the versions of installed charts and supports rollback capabilities. This enables teams to easily revert to a previous version of an application if an update introduces issues or errors. Versioning and rollback functionality provide safety nets for production deployments, allowing teams to quickly recover from failures without impacting users.
- Dependency Management: Helm supports dependency management, allowing charts to declare dependencies on other charts. Helm automatically resolves and installs dependencies when deploying charts, simplifying the management of complex application stacks with multiple components. Dependency

management ensures that all required components are deployed and configured correctly, reducing the risk of misconfigurations and compatibility issues.

- Customization and Templating: Helm templates enable parameterization and customization of Kubernetes manifests. Teams can use templates to dynamically generate configurations based on environment-specific variables, such as namespace names, resource limits, and service endpoints. This facilitates the deployment of applications across different environments (e.g., development, staging, production) with minimal manual intervention.
- Community Ecosystem: Helm has a vibrant ecosystem with a large collection of community-maintained charts available in public repositories, such as the Helm Hub. These charts cover a wide range of applications and services, including databases, web servers, monitoring tools, and more. Leveraging community charts saves time and effort by providing pre-configured solutions for common use cases, accelerating the deployment of applications in production environments.
- Auditing and Compliance: Helm provides visibility into the deployment history and configuration changes of applications through its release management capabilities. Teams can audit and track changes to application configurations over time, ensuring compliance with regulatory requirements and internal policies. Helm's audit trail helps teams maintain a record of changes and troubleshoot issues more effectively in production environments.

Question 3:

How can Helm be integrated into CI/CD pipelines for efficient application deployment?

Answer:

Integrating Helm into CI/CD pipelines can streamline the process of deploying Kubernetes applications efficiently. Here's how Helm can be integrated into CI/CD pipelines:

- Helm Chart Management: Create and maintain Helm charts for your Kubernetes applications. Helm charts should contain templates for Kubernetes manifests, default configurations, dependencies, and metadata.
- Source Control: Store Helm charts and CI/CD pipeline configurations in a version control system like Git. This ensures that changes to charts and pipeline configurations are tracked, versioned, and auditable.
- CI Pipeline Setup:
    - Configure CI pipelines to build, test, and package Helm charts as part of the CI process. This may involve compiling application code, running tests, and generating Helm chart artifacts.
    - Store generated Helm chart artifacts in a repository accessible to the CI pipeline, such as a Docker registry or artifact repository.
- Automated Testing: Include automated tests for Helm charts in the CI pipeline. Helm chart testing can involve linting charts for best practices, validating Kubernetes manifest syntax, and executing integration tests against a Kubernetes cluster.
- Continuous Deployment:
    - Integrate Helm deployment commands into the CD pipeline to deploy Helm charts to Kubernetes clusters automatically.
    - Use Helm commands like helm install, helm upgrade, and helm rollback to manage releases and update deployed applications with new chart versions.
    - Pass configuration values to Helm charts dynamically using environment-specific variables or configuration files.
- Infrastructure Provisioning:
    - Integrate Helm with infrastructure provisioning tools like Terraform or Ansible to automate the provisioning of Kubernetes clusters as part of the CD pipeline.
    - Ensure that Kubernetes clusters are provisioned with the necessary prerequisites (e.g., RBAC roles, namespaces) before deploying applications using Helm.

Question 3:

Describe the different types of Helm charts and their significance in deploying applications.

Answer:

Application charts are used to deploy standalone applications or services onto Kubernetes clusters, while library charts provide reusable components and configurations shared across multiple application charts. By leveraging both types of charts, Helm enables teams to streamline the deployment process, promote consistency, and accelerate the delivery of Kubernetes applications with efficiency and reliability.

Question 4:

What is the role of Tiller in Helm, and how does it contribute to managing releases?

Answer:

Tiller is the server-side component of Helm that interacts with the Kubernetes API server to manage Helm releases. It handles operations such as installing, upgrading, querying, and deleting releases on Kubernetes clusters. Tiller also manages release history, tracks installed charts, and applies configuration changes to deployed releases. However, it's worth noting that Tiller has been deprecated since Helm 3, and Helm 3 no longer requires Tiller for managing releases, improving security and simplifying deployment architecture.

Question 5:

How can Helm be used to manage environment-specific configurations for development, staging, and production environments?

Answer:

- Values Files: Helm allows you to define values files for different environments, such as values-dev.yaml, values-staging.yaml, and values-prod.yaml. These files contain environment-specific configurations, such as service endpoints, resource limits, and feature flags. You can pass these values files to Helm during deployment to customize the configuration of the deployed charts for each environment.
- Override Values: Helm supports overriding individual values from the command line using the --set flag. This enables you to specify environment-specific configurations directly when deploying charts. For example, you can override

the database endpoint or the number of replicas for a deployment using --set flags tailored for each environment.

- Template Conditionals: Helm templates support conditional logic using Go templating. You can use conditionals to include or exclude configuration settings based on the target environment. For example, you can conditionally enable or disable certain features, configure different resource limits, or specify environment-specific secrets based on the environment.
- Release Namespaces: Helm supports deploying releases into different Kubernetes namespaces. You can use separate namespaces for each environment (e.g., dev, staging, prod) to isolate resources and configurations. This allows you to manage environment-specific configurations and prevent interference between different environments sharing the same Kubernetes cluster.
- Helm Hooks: Helm hooks enable you to execute custom scripts or actions during the deployment lifecycle. You can use hooks to perform environment-specific tasks, such as initializing databases, configuring external services, or running integration tests, as part of the deployment process.