Drawbacks of k8s:

1. Static files
2. Consistency – If a dev directly changes the k8s cluster without making changes to yaml files, it will make changes inconsistent
3. K8s doesn’t maintain revision history for us

Helm: Package Manager

Charts: packages

Charts are hosted on chart repository

Charts are required to create k8s resources.

Advantages of Helm

1. Simplify the k8s deployment by extracting out the complexities.
2. Helm maintain revision history
3. Dynamic configuration – We can pass values to deployment.yaml, service.yaml, configmap.yaml and secret.yaml using Values.yaml
4. Consistency- We use helm to do the upgrade on k8s cluster
5. Lifecycle hooks – Actions to perform before any helm commands
6. Security

Helm and K8s use same config files

To overwrite configuration files set KUBECONFIG variable

Helm repo list – Show list of configured repositories

Helm repo add {name} {repo\_url}: To add chart repository

Helm search repo {chart\_name}

Helm repo remove {repo\_name}

Helm status {chart\_name}

**To install same chartname again use another k8s namespace**

**Helm install {chart\_name} –namespace {namespace\_name} –create\_namespace**

Helm list –namespace : list packages install on k8s cluster

Helm uninstall {package\_name} –namespace {namespace\_name} : Uninstal

Helm repo update- update all charts – It will upgrade the revision number

--reuse-values: It will reuse config values while upgrading the chart.

Helm upgrade –install – Do upgrade if it does not exist perform install

Release records: When ever we install or upgrade charts on helm it will store the version history in kubectl get secrets command.

--dry-run = Load chart and its dependencies , substitute values.yaml, validate the schemas and generate k8s files but do not submit those files to k8s to execute and create the objects.

Helm template {chart\_name} –values={path\_to\_values\_yaml\_file}

This will generate k8s file for storing in Code repository and later can be used to generate the objects on the k8s cluster

Helm get notes {chart\_name}: This will display release notes

Helm get values {chart\_name}: This will display values.yaml used while install or upgrading the chart

Helm history {chart\_name}: History of install , upgrade and failures of a particular chart

Helm rollback {chart\_name} {version\_number}

Helm get manifest {chart\_name}: it will get the manifest of the release which was sent to k8s

--keep-history – Keeps all the history related to a particular chart

Helm install {chart\_name} {repo\_path} –wait --timeout 5m10s : Wait and timeout till all the resource are created

Flag used during Upgrade command

--atomic – Will rollback to previous successful release.

--force – During upgrade it will delete the old deployment and create the new deployment

--cleanup-on-failure - this will remove any dangling resource during helm upgrade

Chart Commands:

Helm create {chart\_name} – chart.yaml, values.yaml, charts, templates folder

Chart.yaml – it will have metadata about the chart created

Charts (Folder) – If current chart is using any other chart dependency it will come into this folder.

Templates(folder) – deployment.yaml, service.yaml etc these files will be used to render the k8s manifest

Values.yaml – Default values going in the chart

\_helpers.tpl – IT will have template functions which can be used in deployment.yaml, service.yaml files etc.

Values.yaml – IT will have default values which are used across all the files under templates directory.

Helm package {chart\_folder\_path} -u – Generate tgz file with version with updated dependencies.

.helmignore – It will remove unnecessary files to be removed from helm package

Helm lint – check for syntax or indent issues

**Go Templating Syntax**

* It used to remove any leading or trailing whitespaces

.Values - It is subobject referring to values.yaml file

| - to combine multiple function calls

{{ .Values.my.custom.data | default “testfefault” | upper | quote }}

Nindent – newline with 4 spaces

toYaml – Convert current object to yaml

{{- if .Values.my.flag }}

{{“Output of if” | nident 2}}

{{- else}}

{{“Output of else” | nident 2}}

{{- end}}

To bind list of values to the template

{{- with .Values.my.values}}

{{- toYaml . | nident 2 }}

{{- end}}

**Variable**

{{ $myFlag := “test” }}

Loops

{{- range .Values.my.values}}

- {{. | upper | quote }}

{{- end}}

Dict

{{- range $key,$value .Values.my.values}}

- {{$key}} : {{$value}} | upper | quote }}

{{- end}}

Dependencies are defined in chart.yaml inside dependencies block

Helm dependency update {release\_name}

Tags: are used to send multiple flags

Import-values: fetch child chart values in parent chart

[In summary, **ClusterIP** is used for pod-to-pod communication within the same Kubernetes cluster, while **NodePort** is used for communication between applications within the cluster and external clients outside the cluster 2](https://kodekloud.com/blog/clusterip-nodeport-loadbalancer/)

Values.schema.json- It defines the format , type and values used in values.yaml file

Above file is used in helm lint command

Helm env HELM\_DATA\_HOME – The folder in which helm expects to store a starter template.

Helm create vs apply difference

Helm has inbuilt support for security and provenance using PGP: Pretty Good Privacy

Helm install –verify using GNUPG files

**1. Install Helm**

**Before you start creating a Helm chart, make sure Helm is installed on your local machine. You can install Helm by following the official Helm installation guide.**

**To check if Helm is installed, you can run:**

**bash**

**Copy code**

**helm version**

**2. Create a New Helm Chart**

**You can create a new Helm chart using the helm create command. This will generate a sample chart with the necessary file structure.**

**bash**

**Copy code**

**helm create <chart-name>**

**This command will generate a directory structure like:**

**bash**

**Copy code**

**<chart-name>/**

**Chart.yaml # Metadata about the chart**

**values.yaml # Default values for the chart**

**charts/ # Directory for dependencies (if any)**

**templates/ # Kubernetes manifest templates**

**.helmignore # Files to ignore when packaging the chart**

**3. Understand the Structure**

**Here’s a brief explanation of the files and directories created:**

* **Chart.yaml: Contains metadata about the chart (name, version, description, etc.).**
* **values.yaml: Contains default values that can be overridden during chart installation. These values are used by templates inside the templates/ folder.**
* **templates/: Contains Kubernetes manifest templates (such as deployment.yaml, service.yaml, etc.) that define the resources the chart will create.**
* **charts/: Used to store any dependencies (sub-charts) that your chart relies on.**
* **.helmignore: A file to specify which files should not be packaged with the chart (similar to .gitignore).**

**4. Customize the Chart**

* **Edit Chart.yaml: Provide metadata for your chart, such as the chart’s name, version, description, and maintainers.**

**Example Chart.yaml:**

**yaml**

**Copy code**

**apiVersion: v2**

**name: myapp**

**description: A simple Helm chart for deploying my app**

**version: 0.1.0**

**maintainers:**

**- name: Your Name**

**email: youremail@example.com**

* **Modify values.yaml: This file holds the default values that can be configured by the user during installation. You can define parameters like the number of replicas, image names, and more.**

**Example values.yaml:**

**yaml**

**Copy code**

**replicaCount: 3**

**image:**

**repository: nginx**

**pullPolicy: IfNotPresent**

**tag: "1.14.2"**

**service:**

**type: ClusterIP**

**port: 80**

* **Edit Templates: Customize the templates/ directory by modifying or adding Kubernetes resource files like deployment.yaml, service.yaml, ingress.yaml, etc. Helm templates use Go templating syntax to replace values in the YAML files based on the values.yaml file.**

**Example deployment.yaml:**

**yaml**

**Copy code**

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: {{ .Release.Name }}-deployment**

**labels:**

**app: {{ .Release.Name }}**

**spec:**

**replicas: {{ .Values.replicaCount }}**

**selector:**

**matchLabels:**

**app: {{ .Release.Name }}**

**template:**

**metadata:**

**labels:**

**app: {{ .Release.Name }}**

**spec:**

**containers:**

**- name: {{ .Release.Name }}**

**image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"**

**ports:**

**- containerPort: 80**

**In the above example, {{ .Values.replicaCount }} will be replaced by the value specified in values.yaml (or overridden during installation).**

**5. Package the Chart (Optional)**

**Once you've customized your chart, you can package it into a .tgz file using the helm package command. This will create a chart archive that can be uploaded to a Helm repository or shared directly.**

**bash**

**Copy code**

**helm package <chart-name>**

**This will create a .tgz file that contains your chart, which you can use for deployment or sharing.**

**6. Install the Chart Locally**

**You can install your Helm chart in your Kubernetes cluster for testing purposes using the helm install command.**

**bash**

**Copy code**

**helm install <release-name> <chart-name>/**

**For example:**

**bash**

**Copy code**

**helm install myapp ./myapp**

**This command will deploy the chart to your Kubernetes cluster, using the default values.yaml settings.**

**If you want to override values, you can either modify values.yaml or use the --set flag:**

**bash**

**Copy code**

**helm install myapp ./myapp --set replicaCount=5**

**7. Test the Chart**

**After installation, you should verify that the resources are created and working as expected in your Kubernetes cluster:**

**bash**

**Copy code**

**kubectl get all**

**This will list all the resources created by your Helm chart (such as deployments, pods, services, etc.).**

**8. Upgrade the Chart**

**You can upgrade an existing release of the chart with a new version of the chart or changes to values.yaml using the helm upgrade command.**

**bash**

**Copy code**

**helm upgrade <release-name> <chart-name>**

**For example:**

**bash**

**Copy code**

**helm upgrade myapp ./myapp --set replicaCount=4**

**9. Uninstall the Chart**

**If you want to remove the installed chart, you can use the helm uninstall command:**

**bash**

**Copy code**

**helm uninstall <release-name>**

**For example:**

**bash**

**Copy code**

**helm uninstall myapp**

**10. Share the Chart (Optional)**

**If you want to share your Helm chart with others, you can push it to a Helm repository. For example, to share with a public Helm repository like Artifact Hub or a private Helm repository, you’ll need to follow the specific steps provided by the repository service (such as creating a repository index file and using Helm's helm push command).**

**Lifecycle Summary**

1. **Create the chart using helm create <chart-name>.**
2. **Customize the chart by modifying Chart.yaml, values.yaml, and template files.**
3. **Package the chart (optional) using helm package.**
4. **Install the chart into Kubernetes using helm install.**
5. **Upgrade the chart if needed with helm upgrade.**
6. **Rollback to a previous version if necessary using helm rollback.**
7. **Uninstall the chart and remove the release using helm uninstall.**
8. **Manage the lifecycle by checking the release history and status.**

**1. What is Helm?**

* **Answer**: Helm is a package manager for Kubernetes that helps you manage Kubernetes applications. It allows you to define, install, and upgrade even the most complex Kubernetes applications using Helm charts. Helm provides a way to package Kubernetes resources, making it easier to deploy and manage applications on a Kubernetes cluster.

**2. What are Helm Charts?**

* **Answer**: Helm charts are a collection of Kubernetes YAML templates, configurations, and metadata that define a set of Kubernetes resources. Charts allow you to package, share, and deploy applications and services to Kubernetes in a standardized way. They are used by Helm to deploy applications to the Kubernetes cluster.

**3. What is the structure of a Helm chart?**

* **Answer**: A Helm chart consists of several files and directories:
  + Chart.yaml: Contains metadata about the chart (e.g., name, version, description).
  + values.yaml: Contains default values for the chart’s configuration.
  + templates/: Contains Kubernetes manifest files, which are YAML files with Kubernetes resources (e.g., pods, services, deployments).
  + charts/: Contains any sub-charts (other charts used as dependencies).
  + README.md: Optional documentation about how to use the chart.
  + LICENSE: Optional file for licensing information.

**4. What is the values.yaml file in Helm?**

* **Answer**: The values.yaml file is where the default values for a Helm chart are defined. It allows users to specify configurations that can be customized for a given deployment. When deploying a chart, users can override these values by providing their own custom values file or specifying them via the --set flag.

**5. What are Helm Repositories?**

* **Answer**: Helm repositories are places where Helm charts are stored and shared. They can be public or private and allow you to download charts for installation. Helm repositories are often used to distribute pre-packaged charts for popular applications (e.g., Nginx, MySQL). Popular repositories include the **Helm Hub** and **Bitnami charts**.

**6. How do you install a Helm chart?**

* **Answer**: You can install a Helm chart using the helm install command. Example:

bash

Copy code

helm install <release-name> <chart-name> [flags]

Where:

* <release-name> is the name you want to give the release.
* <chart-name> is the name of the Helm chart you want to install (either local or from a repository).

You can also specify a values.yaml file to customize the installation:

bash

Copy code

helm install <release-name> <chart-name> -f custom-values.yaml

**7. How does Helm handle dependencies?**

* **Answer**: Helm charts can define dependencies on other charts. These dependencies are managed via the charts/directory or by defining them in the Chart.yaml file under the dependencies section. Helm can automatically download and install the required dependencies using the helm dependency update command.

Example of a Chart.yaml with dependencies:

yaml

Copy code

dependencies:

- name: mysql

version: 5.7.1

repository: "https://charts.bitnami.com/bitnami"

**8. What is a Helm release?**

* **Answer**: A **release** is a specific instance of a Helm chart running in a Kubernetes cluster. A release consists of a name, the chart version, and the configuration values applied when the chart was installed. You can have multiple releases of the same chart with different configurations.

**9. How do you upgrade a Helm release?**

* **Answer**: You can upgrade a Helm release using the helm upgrade command. This command allows you to upgrade the chart version or modify the values for an existing release.

Example:

bash

Copy code

helm upgrade <release-name> <chart-name> -f custom-values.yaml

This command upgrades the release to the specified chart and values.

**10. How do you roll back a Helm release?**

* **Answer**: Helm allows you to roll back a release to a previous version using the helm rollback command.

Example:

bash

Copy code

helm rollback <release-name> <revision-number>

This command rolls back the release to the specified revision (Helm keeps a history of releases and revisions).

**11. What are Helm hooks?**

* **Answer**: Helm hooks are special resources in the Helm chart that are executed at certain points during the lifecycle of a release. Hooks can be used for tasks like database migrations, pre-installation or post-installation scripts, and other pre/post upgrade tasks.

Example of a hook in a chart template:

yaml

Copy code

apiVersion: batch/v1

kind: Job

metadata:

name: my-job

annotations:

"helm.sh/hook": post-install

spec:

...

**12. What is the helm uninstall command?**

* **Answer**: The helm uninstall command is used to remove a Helm release from the cluster. This command will delete all the resources associated with the release.

Example:

bash

Copy code

helm uninstall <release-name>

**13. What is helm list?**

* **Answer**: The helm list command is used to display all the releases that are currently installed in the Kubernetes cluster. It shows details like the release name, namespace, chart version, and status.

Example:

bash

Copy code

helm list

**14. What is the difference between helm install and helm upgrade?**

* **Answer**:
  + helm install: Installs a new release of a chart into a Kubernetes cluster.
  + helm upgrade: Upgrades an existing release to a new version of the chart or applies new configurations.

**15. How do you manage values in Helm?**

* **Answer**: Values in Helm can be managed in multiple ways:
  + **values.yaml**: Default values defined in the chart.
  + **Override with a custom values file**: Use the -f flag to pass a custom values file.
  + **Override with --set**: Set individual values directly from the command line, like --set key=value.

**16. How do you check the history of a Helm release?**

* **Answer**: You can view the history of a Helm release with the helm history command. This shows the revision history and status of the release.

Example:

bash

Copy code

helm history <release-name>

**17. What is helm repo and how do you use it?**

* **Answer**: helm repo is a command that manages Helm chart repositories. It allows you to add, list, and remove repositories for Helm charts.

Examples:

* Add a new Helm repository:

bash

Copy code

helm repo add <repo-name> <repo-url>

* List all added repositories:

bash

Copy code

helm repo list

**18. What is the difference between Helm 2 and Helm 3?**

* **Answer**: Helm 3 is the latest version, and it introduced several changes from Helm 2:
  + Helm 3 removed the need for Tiller (the server-side component) which was present in Helm 2, improving security.
  + Helm 3 introduces the concept of **Helm 3 client-only** with no Tiller service running in the cluster.
  + Helm 3 has improved security with the removal of Tiller, as it no longer requires cluster-wide permissions.

**Bonus Questions (Advanced)**

**19. What is Helm Chart Testing and how do you test a chart?**

* **Answer**: Helm provides a framework for testing charts, allowing you to validate that your chart installs and configures properly. You can use **helm unittest** or integrate with testing frameworks like **Kuttl** or **ChartMuseum**to test charts.

**20. How can you create your own Helm charts?**

* **Answer**: You can create a Helm chart using the helm create <chart-name> command. This generates the necessary chart structure with sample files. You can then modify the templates and values as needed for your application.