Kubernetes Persistency

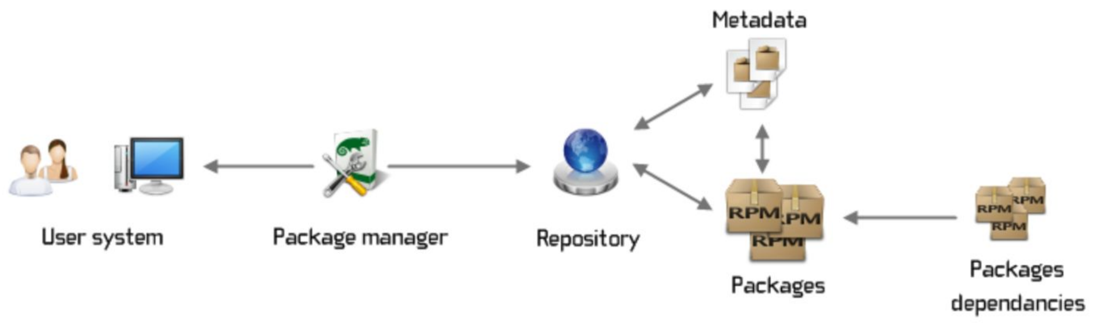
- Managing storage is a distinct problem from managing compute.
- The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed.
- To do this we introduce two new API resources:
  - PersistentVolume
  - PersistentVolumeClaim.
- A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator.
- It is a resource in the cluster just like a node is a cluster resource.
- PVs are volume plugins like Volumes, but have a lifecycle independent of any individual pod that uses the PV.
- PersistentVolume captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.
- A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a pod.
- Pods consume node resources and PVCs consume PV resources.
- Pods can request specific levels of resources (CPU and Memory).
- Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).
- While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems.
- Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented. For these needs there is the StorageClass resource.

- Persistent Volumes can be provisioned in two ways:
  - Static - A cluster administrator creates a number of PVs. They carry the details of the real storage which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.
  - Dynamic - When none of the static PVs the administrator created matches a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC.
- A user creates, or has already created in the case of dynamic provisioning, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes.
- A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together.

- If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC.
- Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested.
- ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable.
- ConfigMap resource holds key-value pairs of configuration data that can be consumed in pods or used to store configuration data for system components such as controllers.
- ConfigMap designed to more conveniently support working with strings that do not contain sensitive information.
- Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping.
- Claims will remain unbound indefinitely if a matching volume does not exist.
- ConfigMaps are not intended to act as a replacement for a properties file.
- ConfigMaps are intended to act as a reference to multiple properties files.
- You can think of them as way to represent something similar to the /etc directory, and the files within, on a Linux computer. One example of this model is creating Kubernetes Volumes from ConfigMaps, where each data item in the ConfigMap becomes a new file.

Helm, the Package Manager for Kubernetes

- A package management system is a collection of tools that provides a consistent method of installing, upgrading and removing software on your system.
- Most every programming language and operating system has its own package manager to help with the installation and maintenance of software. Many of the package managers you may already be familiar with, such as Debian's apt, or Python's pip.
- Software is distributed through Packages that are linked to metadata which contain additional information such as a description of the software purpose and a list of dependencies necessary for the software to run properly.
- Packages are archives of files that include all the files making up a piece of software (such as an application itself, shared libraries, development packages containing files needed to build software against a library, ...) and, eventually, instructions on the way to make them work.



Helm Package Manager

- Deploying applications to Kubernetes – the powerful and popular container-orchestration system – can be complex.
- Setting up a single application can involve creating multiple interdependent Kubernetes resources – such as pods, services, deployments, and replicasets – each requiring you to write a detailed YAML manifest file.
- Helm is a package manager for Kubernetes that allows developers and operators to more easily package, configure, and deploy applications and services onto Kubernetes clusters.
- Helm is now an official Kubernetes project and is part of the **Cloud Native Computing Foundation**, a non-profit that supports open source projects in and around the Kubernetes ecosystem.

Helm can:

- Install software.
- Automatically install software dependencies.
- Upgrade software.
- Configure software deployments.
- Fetch software packages from repositories.

Helm provides this functionality through the following components:

- A command line tool, helm, which provides the user interface to all Helm functionality.
- A companion server component, tiller, that runs on your Kubernetes cluster, listens for commands from helm, and handles the configuration and deployment of software releases on the cluster.
- The Helm packaging format, called charts.
- An official curated charts repository with prepackaged charts for popular open-source software projects.

Helm Charts
Helm packages are called charts, and they consist of a few YAML configuration files and some templates that are rendered into Kubernetes manifest files. Here is the basic directory structure of a chart:

```
package-name/
  charts/
  templates/
  Chart.yaml
  LICENSE
  README.md
  requirements.yaml
  values.yaml
```

- **charts/**: Manually managed chart dependencies can be placed in this directory, though it is typically better to use requirements.yaml to dynamically link dependencies.
- **templates/**: This directory contains template files that are combined with configuration values (from values.yaml and the command line) and rendered into Kubernetes manifests. The templates use the Go programming language's template format.
- **Chart.yaml**: A YAML file with metadata about the chart, such as chart name and version, maintainer information, a relevant website, and search keywords.
- **requirements.yaml**: A YAML file that lists the chart's dependencies.
- **values.yaml**: A YAML file of default configuration values for the chart.

- The helm command can install a chart from a local directory, or from a **.tar.gz** packaged version of this directory structure.
- These packaged charts can also be automatically downloaded and installed from chart repositories or repos.
- A Helm chart repo is a simple HTTP site that serves an index.yaml file and .tar.gz packaged charts. The helm command has subcommands available to help package charts and create the required index.yaml file. These files can be served by any web server, object storage service, or a static site host such as GitHub Pages.

Helm Predefined Values

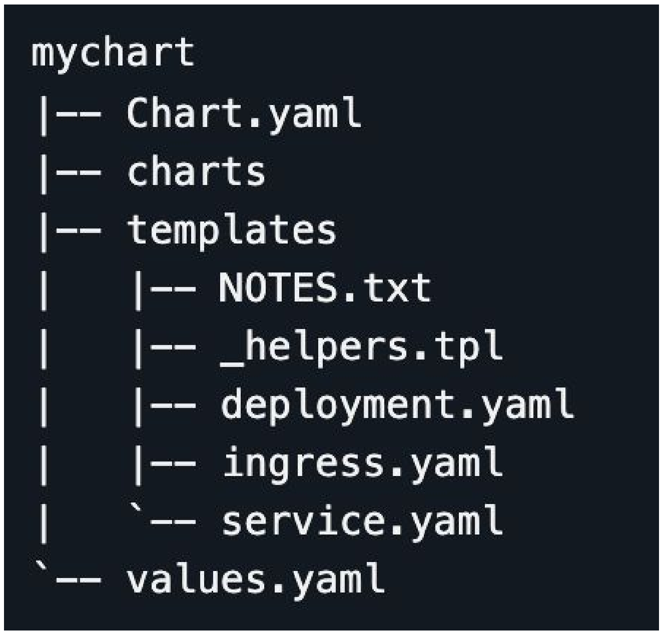
- Values that are supplied via a values.yaml file (or via the --set flag) are accessible from the .Values object in a template.
- A values file is formatted in YAML.
- A chart may include a default values.yaml file.
- The Helm install command allows a user to override values by supplying additional YAML values as we will see in the upcoming demo.

Helm Installation and Demo

1. Follow instructions under k8s-demo/helm/helm-init-commands.txt
   a. curl -L https://git.io/get_helm.sh | bash
   b. helm init # setup helm with our cluster
   c. helm repo update # sync all helm charts info
   d. helm repo list
   e. helm list
2. Follow instructions under k8s-demo/helm/deploy-helm-chart.txt
3. Now let’s review the values.yaml
4. But how all of this changes are set to the application?

Creating Helm Chart

The best way to get started with a new chart is to use the helm create command to scaffold out an example we can build on. Use this command to create a new chart.
helm create mychart

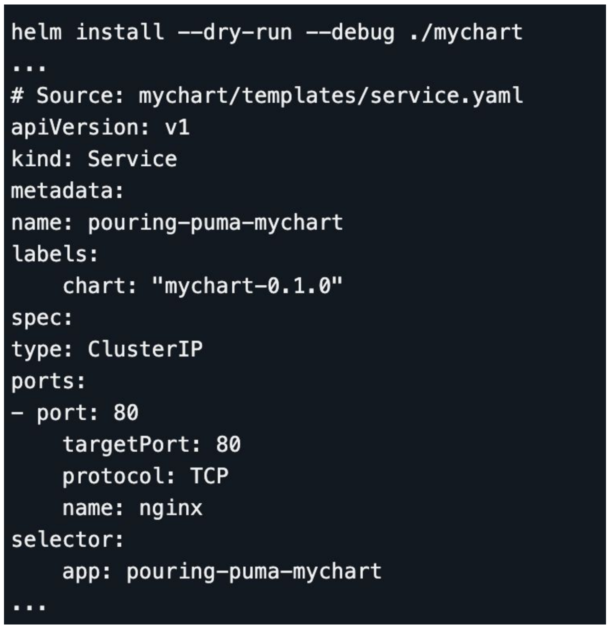


```
mychart
|-- Chart.yaml
|-- charts
|-- templates
|   |-- NOTES.txt
|   |-- _helpers.tpl
|   |-- deployment.yaml
|   |-- ingress.yaml
|   `-- service.yaml
`-- values.yaml
```

Templates

- The most important piece of the puzzle is the templates/ directory.
- This is where Helm finds the YAML definitions for your Services, Deployments and other Kubernetes objects.
- If you already have definitions for your application, all you need to do is replace the generated YAML files for your own.
- What you end up with is a working chart that can be deployed using the helm install command.

This is a basic Service definition using templating. When deploying the chart, Helm will generate a definition that will look a lot more like a valid Service. We can do a dry-run of a helm install and enable debug to inspect the generated definitions:


```
helm install --dry-run --debug ./mychart
...
# Source: mychart/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
name: pouring-puma-mychart
labels:
    chart: "mychart-0.1.0"
spec:
type: ClusterIP
ports:
- port: 80
    targetPort: 80
    protocol: TCP
    name: nginx
selector:
    app: pouring-puma-mychart
...
```

Values.yaml

- The template makes use of the Helm-specific objects .Chart and .Values.
- .Chart provides metadata about the chart to your definitions such as the name, or version.
- .Values object is a key element of Helm charts, used to expose configuration that can be set at the time of deployment.
- The defaults for this object are defined in the values.yaml file.

Developing Helm Chart

- follow the steps at helm/dev-and-deploy-chart.txt
    a. helm create mychart
    b. helm install ./mychart --dry-run --debug --name mychart
    c. helm install ./mychart --name mychart
    d. helm package ./mychart
    e. helm upgrade --install mychart mychart-0.1.0.tgz --set replicaCount=3
    f. kubectl get pods

Helm Chart Repository

- A chart repository is an HTTP server that houses an index.yaml file and optionally some packaged charts.
- When you're ready to share your charts, the preferred way to do so is by uploading them to a chart repository.
- Because a chart repository can be any HTTP server that can serve YAML and tar files and can answer GET requests, you have a variaty of options when it comes down to hosting your own chart repository.
- For example, you can use a Google Cloud Storage (GCS) bucket, Amazon S3 bucket, Github Pages, or even create your own web server.
- ChartMuseum - ChartMuseum is an open-source, easy to deploy, Helm Chart Repository server.
- To deploy a chart museum repository on our minikube cluster go to helm folder and run deploy-repository.txt commands