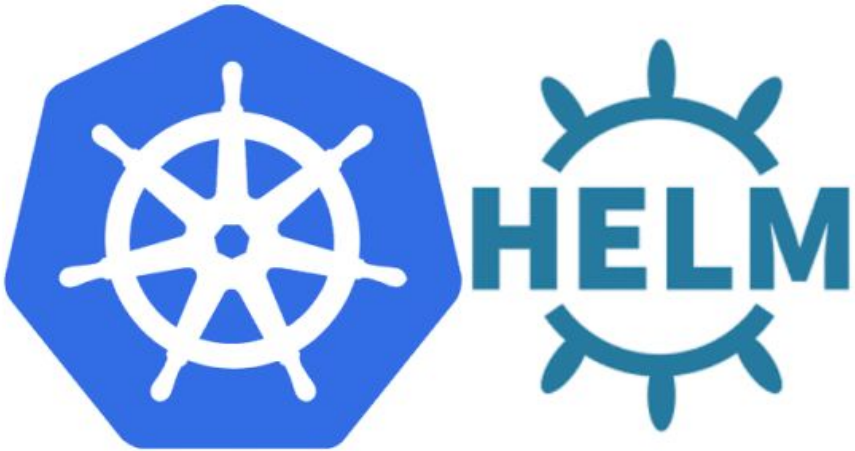
Kubernetes For DevOps

Lev Epshtein


HELM

Kubernetes Helm

Helm is a tool for managing Kubernetes packages called *charts*. Helm can do the following:

- Create new charts from scratch
- Package charts into chart archive (tgz) files
- Interact with chart repositories where charts are stored
- Install and uninstall charts into an existing Kubernetes cluster
- Manage the release cycle of charts that have been installed with Helm

Three big concepts

A *Chart* is a Helm package. It contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster. Think of it like the Kubernetes equivalent of a Homebrew formula, an Apt dpkg, or a Yum RPM file.

A *Repository* is the place where charts can be collected and shared. It's like Perl's CPAN archive or the Fedora Package Database, but for Kubernetes packages.

A *Release* is an instance of a chart running in a Kubernetes cluster. One chart can often be installed many times into the same cluster. And each time it is installed, a new *release* is created. Consider a MySQL chart. If you want two databases running in your cluster, you can install that chart twice. Each one will have its own *release*, which will in turn have its own *release name*.

Define a Chart

A chart is organized as a collection of files inside of a directory. The directory name is the name of the chart.

Let's create "Hello, World" chart:

```
$ mkdir ./hello-world
$ cd ./hello-world
$ cat <<'EOF' > ./Chart.yaml
name: hello-world
version: 1.0.0
EOF
```

A chart must define templates used to generate Kubernetes manifests, e.g.

```
$ mkdir ./templates
$ cat <<'EOF' > ./templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: hello-world
spec:
 selector:
   matchLabels:
     app: hello-world
 replicas: 1
 template:
   metadata:
     labels:
       app: hello-world
   spec:
     containers:
       - name: hello-world
         image: gcr.io/google-samples/node-hello:1.0
         ports:
           - containerPort: 8080
             protocol: TCP
EOF
```

```
$ cat <<'EOF' > ./templates/service.yaml
apiVersion: v1
kind: Service
metadata:
 name: hello-world
spec:
 type: NodePort
 ports:
 - port: 8080
   targetPort: 8080
   protocol: TCP
 selector:
   app: hello-world
EOF


$ microk8s.helm install .
```

```
$ microk8s.kubectl get po,svc

$ microk8s.helm ls

$ microk8s.helm status RELEASE_NAME

$ microk8s.helm delete RELEASE_NAME

$ microk8s.helm ls --deleted

$ microk8s.helm rollback RELEASE_NAME REVISION_NUMBER

$ microk8s.helm delete --purge RELEASE_NAME
```

Configuring releases

Helm Chart templates are written in the Go template language, with the addition of 50 or so add-on template functions from the Sprig library.

Values for the templates are supplied using values.yaml file

Values that are supplied via a values.yaml file are accessible from the .Values object in a template.

```
$ cat <<'EOF' > ./templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: hello-world
spec:
 selector:
   matchLabels:
     app: hello-world
 replicas: 1
 template:
   metadata:
     labels:
       app: hello-world
   spec:
     containers:
       - name: hello-world
         image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
         ports:
           - containerPort: 8080
             protocol: TCP
EOF
```

Values in the `values.yaml` can be overwritten at the time of making the release using `--values YAML_FILE_PATH` or `--set key1=value1,key2=value2` parameters, e.g.

```
$ helm install --set image.tag='latest' .
```

Values provided using `--values` parameter are merged with values defined in values.yaml file and values provided using `--set` parameter are merged with the resulting values, i.e. `--set` overwrites provided values of `--value`, --value overwrites provided values of `values.yaml` .

Debugging

Using templates to generate the manifests require to be able to preview the result. Use `--dry-run --debug` options to print the values and the resulting manifests without deploying the release

```
$ helm install . --dry-run --debug --set image.tag=latest
```

Using predefined values

In addition to user supplied values, there is a number of predefined values:

- *.Release* is used to refer to the resulting release values, e.g. *.Release.Name*, *.Release.Time* .
- *.Chart* is used to refer to the *Chart.yaml* configuration.
- *.Files* is used to refer to the files in the chart directory.

Example in: *../k8s-3/hello-world3*

Partials

You have probably noticed a repeating pattern in the templates:

```
app: {{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
version: {{ .Chart.Version }}
release: {{ .Release.Name }}
```

We are using the same labels for all resources.

Furthermore, our resource name is a lengthy expression:

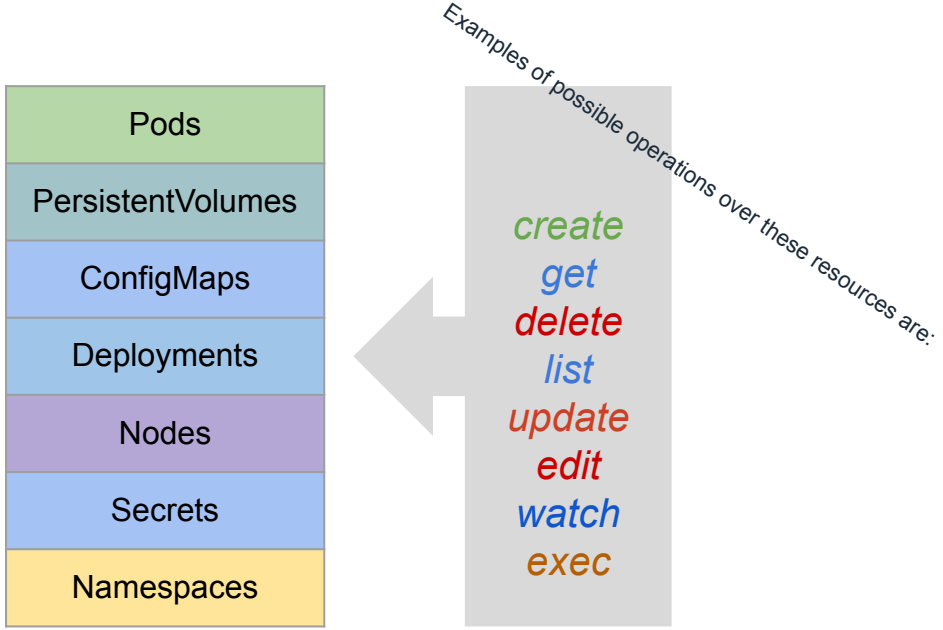
```
{{ printf "%s-%s" .Release.Name .Chart.Name | trunc 63 }}
```

Files in ./templates directory that start with _ are not considered Kubernetes manifests. The rendered version of these files are not sent to Kubernetes.

RBAC API Objects

One basic Kubernetes feature is that all its resources are modeled API objects, which allow CRUD (Create, Read, Update, Delete) operations.

Examples of resources are:



At a higher level, resources are associated with API Groups (for example, Pods belong to the *core* API group whereas Deployments belong to the *apps* API group). For more information about all available resources, operations, and API groups, check the Official Kubernetes API Reference.

To manage RBAC in Kubernetes, apart from resources and operations, we need the following elements:

- Rules
- Roles and ClusterRoles
- Subjects
- RoleBindigs

Follow tutorial:
https://docs.bitnami.com/kubernetes/how-to/configure-rbac-in-your-kubernetes-cluster/

Istio

What is Istio?

- Open source service mesh that layers transparently onto existing distributed applications.
- Platform, including APIs that let it integrate into any logging platform, or telemetry or policy system.
- Istio’s diverse feature set lets you successfully, and efficiently, run a distributed microservice architecture, and provides a uniform way to secure, connect, and monitor microservices.

What is service mesh?

The term service mesh is used to describe the network of microservices that make up such applications and the interactions between them. As a service mesh grows in size and complexity, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring. A service mesh also often has more complex operational requirements, like A/B testing, canary rollouts, rate limiting, access control, and end-to-end authentication.

Why to use Istio?

- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas.
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress.
- Secure service-to-service communication in a cluster with strong identity-based authentication and authorization.

Core feature

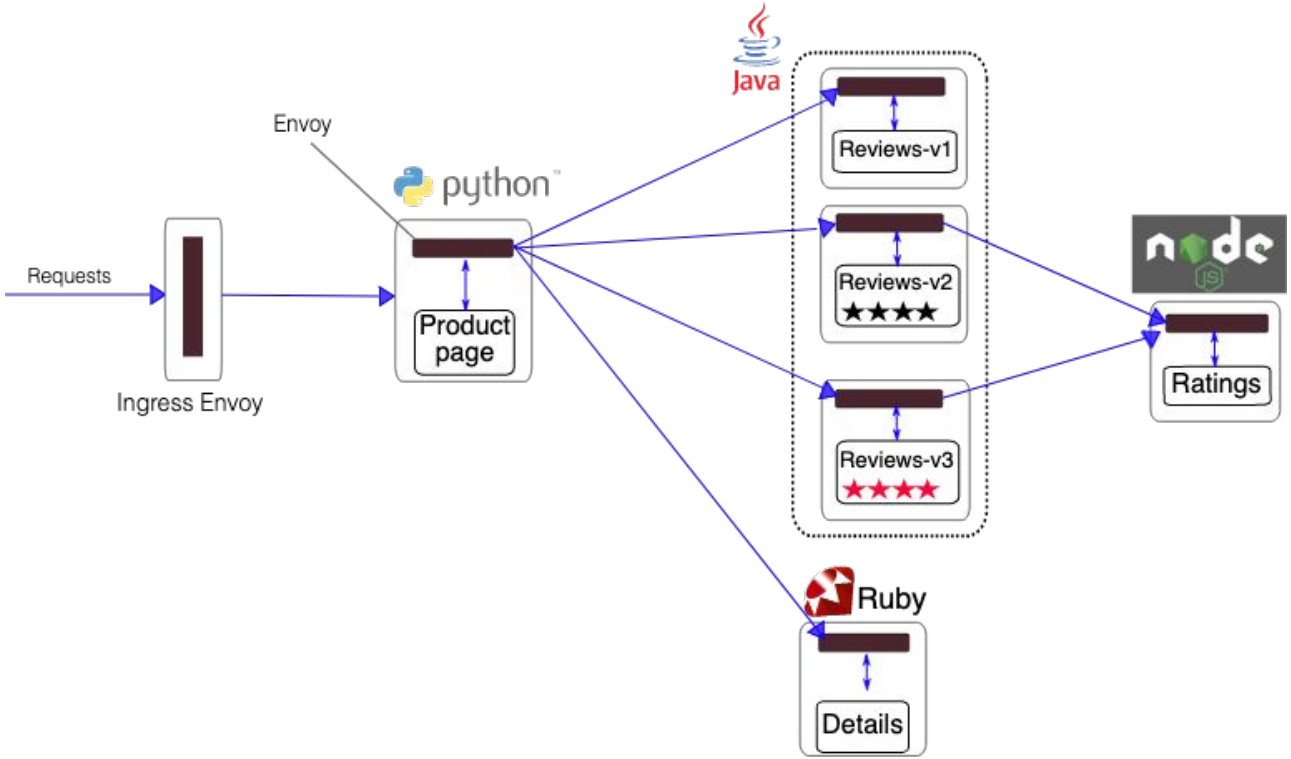
- **Traffic management**
- **Security**
- **Policies**
- **Observability**

Mastering Istio

Quickest and Easiest way to mastering Istio:

- Documentation Example and Tasks

https://istio.io/docs/examples/bookinfo/


Java
Envoy
python
Reviews-v1
Reviews-v2
Requests
Product page
Ratings
Reviews-v3
Ingress Envoy
Ruby
Details