Packaging and Shipping Operators

Learn how to package and ship Kubernetes operators.

We'll cover the following

- Overview
- Preparing raw manifests
- Shipping with Helm charts
- Conclusion

Overview

Now that we're familiar with how to develop an operator with best practices, let's talk about the next step: packaging and shipping our operator. When we're ready to ship or release our operator, we need to provide a way to install it, even if we're the only user.

The most straightforward manner for delivering our operator is to provide raw manifests, so users can directly apply them to clusters. Let's start with doing this.

Preparing raw manifests

In Kubernetes, operators can run as Pods in the cluster. This is a good way to make sure operators are running with high availability. We need to package all the respective artifacts, which are typically written in YAML. Among these artifacts, we usually have a `Deployment` with multiple replicas, where we define how to run the operator with environment variables, command arguments, configurations, etc. Apart from this, we declare CRDs to describe what our custom APIs look like. Along with these files, we provide security-related resources, such as service accounts and corresponding RBAC rules.

Let's start with a concrete example. Assume we have the following YAML manifest, declaring the operator we'd like users to install. A dedicated namespace is usually necessary, where we can define all the components and configurations. Here, we use a namespace called `pwk-system`, where a ServiceAccount `pwk-demo-operator` is declared as well. In the `pwk-demo-operator`, we specify `3` replicas, the ServiceAccount name, a container running with environment variables, command arguments, and so on.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: pwk-system

---
kind: ServiceAccount
```

```yaml
kind: ServiceAccount
apiVersion: v1
metadata:
  name: pwk-demo-operator
  namespace: pwk-system

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pwk-demo-operator
  namespace: pwk-system
  labels:
    app: pwk-demo-operator
spec:
  replicas: 3
  selector:
    matchLabels:
      app: pwk-demo-operator
  template:
    metadata:
      labels:
        app: pwk-demo-operator
    spec:
      serviceAccountName: pwk-demo-operator
```

The Deployment YAML file of our operator

We also need to grant appropriate permissions to our operator so that it can access the desired resources and take actions. With the RBAC rules below, we assign permissions for our operator to perform all kinds of actions on the resources `foos` in the group `pwk.educative.io`. We could also bind extra RBAC rules if needed.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pwk-demo-operator
rules:
  - apiGroups: [ "pwk.educative.io" ]
    resources: [ "foos" ]
    verbs: [ "*" ]
  - nonResourceURLs: [ "*" ]
    verbs: [ "get" ]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pwk-demo-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pwk-demo-operator
subjects:
  - kind: ServiceAccount
    name: pwk-demo-operator
    namespace: pwk-system
```

RBAC rules for our operator

The manifest files above are straightforward and easy to deploy to clusters. However, they have shortcomings; all the values in the manifest files are fixed. When we try to bump the container image or change the configurations, we need to update the files. Users will likely overwrite those manifest files based on their own preferences.

To overcome these limitations of raw, static YAML manifest files, we can use Helm charts to ship our operator.

Shipping with Helm charts

More and more operators are being shipped with Helm charts. **Helm** enables us to install and upgrade these parameterized charts within a minute. Below is an example `Deployment` for our operator. As we can see, some variables are encoded in the Go template form `{{ .Some.Value.Here}}`. By specifying the override values when running `helm install`, we can apply our customized configurations, such as container image, replicas, command arguments, Pod labels, etc.

```
apiVersion: {{ include "common.capabilities.deployment.apiVersion" . }}
kind: Deployment
metadata:
  name: {{ include "common.names.fullname" . }}
  namespace: {{ .Release.Namespace }}
  {{- if .Values.commonAnnotations }}
  annotations: {{- include "common.tplvalues.render" ( dict "value" .Values.commonAnnotations "context"
  {{- end }}
  labels: {{- include "common.labels.standard" . | nindent 4}}
    app: {{ include "common.names.fullname" . }}
    {{- if .Values.commonLabels }}
    {{- include "common.tplvalues.render" ( dict "value" .Values.commonLabels "context" $ ) | nindent 4
    {{- end }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels: {{- include "common.labels.matchLabels" . | nindent 6 }}
      app: {{ include "common.names.fullname" . }}
  template:
    metadata:
      labels: {{- include "common.labels.standard" . | nindent 8 }}
        app: {{ include "common.names.fullname" . }}
    spec:
      {{- include "common.images.pullSecrets" ( dict "images" (list .Values.image) "global" .Values.glob
      {{- if .Values.affinity }}
      affinity: {{- include "common.tplvalues.render" (dict "value" .Values.affinity "context" $) | ninc
      {{- else }}
      affinity:
        podAffinity: {{- include "common.affinities.pods" (dict "type" .Values.podAffinityPreset "contex
        podAntiAffinity: {{- include "common.affinities.pods" (dict "type" .Values.podAntiAffinityPreset
        nodeAffinity: {{- include "common.affinities.nodes" (dict "type" .Values.nodeAffinityPreset.typ
```

Deployment template for our operator

Creating a Helm chart is very easy. We can run the command `helm create` to create a scaffold chart. The following command creates a scaffold chart for our operator `pwk-demo-operator`:

```
helm create pwk-demo-operator
```

Creates a scaffold Helm chart

After that, we'll get a folder named `pwk-demo-operator` in the current folder. Let's take a look by running the following command in the terminal above:

```
tree ./
```

List the contents in the scaffold directory

The output will be as follows:

```
./
`-- pwk-demo-operator
    |-- charts
    |-- Chart.yaml
    |-- templates
    |   |-- deployment.yaml
    |   |-- _helpers.tpl
    |   |-- hpa.yaml
    |   |-- ingress.yaml
    |   |-- NOTES.txt
    |   |-- serviceaccount.yaml
    |   |-- service.yaml
    |   `-- tests
    |       `-- test-connection.yaml
    `-- values.yaml

4 directories, 10 files
```

The contents of the scaffold directory

In the folder, there are a group of template YAML files. We can directly modify these files to add our own settings.

Conclusion

Helm is a popular package manager for Kubernetes. It provides an easy way to release applications built for Kubernetes, such as operators.

← Back

Best Practices for Building Kubernetes Operators

Next →

Quiz on Kubernetes Operators