Passing configs to Operators through Kustomize

Operators

Operators enable users to create, configure and manage, not just the stateless but also stateful, Kubernetes applications. An operator has its custom controller watching the custom resources specifically defined for the applications. Hence an operator mainly consists of Kubernetes CustomResourceDefinitions (CRDs) and Controller logic.

With operators, management of complex applications and services becomes easy but writing an operator is not simple until the `Operator SDK` was introduced as part of the Operator Framework. Operator SDK enables users to bootstrap a new project fast as well as provides rich high level APIs and extensions for writing operational logics. It provides three types of workflows for users to write operators in Go, Ansible and Helm.

For users familiar with Ansible, creating an Ansible type operator with Operator SDK is simple and fast. The scaffolding and code generation are taken care by the SDK. The reconciling logic for the application is driven by the ansible playbooks and roles, written by users. The operator deployment manifests may be modified to suit the specific operator and application. To configure an operator or the application managed by the operator, users can pass the configurations as environment variables in the `operator.yaml` file generated by the Operator SDK.

In this tutorial, we run Operator SDK CLI to create an Ansible type operator, running following command.

```
operator-sdk new hello-world --api-version=ibm.com/v1alpha1 --kind=Hello --type=ansible
```

Kustomize

Kustomize is Kubernetes native configuration management. It offers a template-free way to customize application configuration using plain YAML files. `kustomize` can be installed as a standalone binary or use with `kubectl` as `apply -k` command.

To run `kustomize` with an application project, the project should have directory structure like following:

```
.
├── base
|   ├── crd.yaml
|   ├── kustomization.yaml
|   ├── operator.yaml
|   ├── role.yaml
|   ├── role_binding.yaml
|   └── service_account.yaml
└── overlays
    ├── production
    |   ├── config-map.yaml
    |   └── kustomization.yaml
    └── staging
        ├── config-map.yaml
        └── kustomization.yaml
```

There are one `kustomization.yaml` file and other resource files in the `base` directory. This is the common configuration for the application. With following command

```
kustomize build base > base.yaml
```

the generated YAML file can be applied to a cluster.

```
kubectl apply -f base.yaml
```

To manage variants of configuration, use `overlays` to modify/patch/merge the common `base`. In each overlay directory, there are one `kustomization.yaml` file and other resource files. To generate the final deployment YAML file with a specific overlay, run

```
kustomize build overlays/production > production.yaml
```

the generated `production.yaml` contains all the resources from `base` as well as any configuration changes in the `production` overlay. It can then be applied to a cluster.

Passing configs to an oeprator with kustomize

In real world, an operator may be deployed to different cluster environments, such as development, staging and production. This means that an operator requires different configurations. For example, an operator may be deployed to different namespace and granted different authorization. Furthermore, the application managed by an operator may also take different configurations.

Here, we are passing the configuration to an operator using the `env` session in the operator `Deployment` YAML file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 1
  selector:
    matchLabels:
      name: hello-world
  template:
    metadata:
      labels:
        name: hello-world
    spec:
      serviceAccountName: hello-world
      containers:
...
          env:
            - name: WATCH_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: OPERATOR_NAME
              value: "hello-world"
            - name: DEPLOY_ENV
              valueFrom:
                configMapKeyRef:
                  name: install-config
                  key: DEPLOY_ENV
...
```

The `DEPLOY_ENV` is a configuration to be modified among different cluster deployment. We are using kustomize to change the configuration for different deployments.

To achieve this, the `kustomization.yaml` in the `base` directory generates the ConfigMap `install-config` which contains just one config as follow:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- crd.yaml
- service_account.yaml
- role.yaml
- role_binding.yaml
- operator.yaml
commonLabels:
  kustomize.component: hello-world
images:
- name: hello-world
  newName: adrian555/hello-world
  newTag: v0.0.1
- name: hello-op
  newName: adrian555/hello-op
  newTag: v0.0.1
configMapGenerator:
- name: install-config
  literals:
    - DEPLOY_ENV="base"
```

Then in each overlay, we update the `DEPLOY_ENV` with different value

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: install-config
data:
  DEPLOY_ENV: "production"
```

and patch it with the `kustomization.yaml` file

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../../base
patchesStrategicMerge:
- config-map.yaml
```

And that is all we need to do.

Codes for the tutorial are kept in this repo.

The `hello-image` directory has the Dockerfile to build the image for `Hello World` sample application/service, taken from Kubernetes tutorials.

The `hello-world` directory has the operator code generated by Operator SDK. The reconciling logic is the ansible role in `hello-world/roles/hello` directory. Note that the `DEPLOY_ENV` configuration is eventually passed on to the application through the operator, in `hello-world.yaml.j2` template file.

The `hello-kustomize` directory contains the base and overlays YAML files for kustomize. The files in `resources` session of `base/kustomization.yaml` file are copied from the `hello-world` 's `deploy` directory since they are required for deploying the operator.

To deploy the operator with `staging` configuration, run following command

```
pushd hello-kustomize/overlays/staging
kustomize build | kubectl apply -f -
popd
```

Note: replace `overlays/staging` with `overlays/production` or `base` to generate the specific deployment for different environment.

Now the operator should be up and running

```
kubectl get pods
## NAME                                READY   STATUS    RESTARTS   AGE
## hello-world-694cc7b887-lnlcl        2/2     Running   0          20m
```

To install the application managed by this operator, apply a CustomResource

```
pushd hello-world/deploy/crds
kubectl apply -f ibm_v1alpha1_hello_cr.yaml
popd
```

wait until the `hello-world` service is up and running

```
kubectl get svc
## NAME                  TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)
  AGE
## hello-world           LoadBalancer   172.21.204.30   169.62.90.107   80:31308/TCP
  13m
```

Once the `hello-world` service is running, ping the service to view the output:

```
curl http://169.62.90.107:31308
## Hello staging!
```

The service returns the specified config for each kustomize base or overlay setting.

Final words

kustomize is a powerful and native Kubernetes configuration tool. It simplifies the configuration task and so enhances configurable operators. This tutorial only demonstrates a single configuration with container env, but in fact, with kustomize, users can also patch configurations of other forms, such as json patch, runtime data with vars, etc.

Operators are effective and efficient approach for managing applications. Operators are also Kubernetes applications. To manage them, as part of Operator Framework, Operator Lifecycle Manager (OLM) was introduced. OLM takes care the lifecycle of operators including the updates to the operators and their resources. OLM also becomes part of Openshift 4.x Container Platform. We will explain how to combine the power of OLM and kustomize and use them in managing applications in the coming article.