Deploy with kustomize

Kustomize is a template-free, Kubernetes native way to customize application configuration. Kustomize is available in kubectl by running `kubectl apply -k`. Kustomize provides a declarative approach to configuration management that you can use with a variety of deployment tools for Kubernetes.

Motivation for use with Redis Enterprise

Our operator is deployed onto your target Kubernetes cluster via a bundle or automation tools like OpenShift’s OLM. The standard practice is to deploy this bundle without any configuration changes as the specific settings are part of how our product has been designed. You are unlikely to need any configuration changes for the operator itself.

The need for configuration management arises in the use of the operator via the Custom Resources defined for Redis Enterprise clusters and databases (see the documentation for the cluster CR and database CR). You will likely want to set different parameters for different applications while having consistency for company policies. At the same time, you’ll want to manage the differences in deployment topologies for development, QA, and production clusters and databases.

For example, production clusters will likely require more resources and greater limits than development or testing systems. In addition, your secrets, passwords, and other security settings are likely to differ across deployment environments. Kustomize makes it easier to handle these differences between environments.

A simple example

Kustomize works on the building variants from a hierarchy of configurations. In the simplest setup, there is one “base” directory containing all of the common resources. For each variant, there are different directories, each with instructions on how to produce the variant.

For example, say we want to control the limits and requests for a node so that they are different values for development and production. To start, we must create a “base” directory that contains a default configuration for a simple cluster:

```bash
mkdir -p base
cat << EOF > base/cluster.yaml
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: test
spec:
  nodes: 3
  username: "app@example.org"
EOF
```

We then need to configure the “base” directory to be recognized by kustomize. The simplest configuration lists only the resources:

```bash
cat << EOF > base/kustomization.yaml
resources:
  - cluster.yaml
EOF
```

Now we can create a variant for development that tunes down the CPU requirements and also creates a single-node cluster while changing the namespace:
mkdir -p dev
cat << EOF > dev/cluster.yaml
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: test
spec:
  nodes: 1
  redisEnterpriseNodeResources:
    limits:
      cpu: 1
      memory: 4Gi
    requests:
      cpu: 1
      memory: 4Gi
EOF

In this example, the cluster resource must repeat the prolog metadata so that kustomize can match our resources. At minimum, we need apiVersion, kind, and metadata.name.

As before, we need to tell kustomize about our resources using a kustomization.yaml file. In this case, we need to tell kustomize what to do with our variant of cluster.yaml. Specifically, we want to patch the base cluster.yaml by strategically merging properties:

cat << EOF > dev/kustomization.yaml
bases:
  - ../base

namespace: dev

patchesStrategicMerge:
  - cluster.yaml
EOF

For a QA deployment, we might need more resources:
mkdir -p qa
cat << EOF > qa/cluster.yaml
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: test
spec:
  nodes: 5
  redisEnterpriseNodeResources:
    limits:
      cpu: 2
      memory: 25Gi
    requests:
      cpu: 2
      memory: 25Gi
EOF
cat << EOF > qa/kustomization.yaml
bases:
- ../base
namespace: qa
patchesStrategicMerge:
- cluster.yaml
EOF

We can now test our “dev” customization without applying it to the cluster by running the following:

```
kubectl kustomize dev
```

and the output should look something like this:

```
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: test
  namespace: dev
spec:
  nodes: 1
  redisEnterpriseNodeResources:
    limits:
      cpu: 1
      memory: 4Gi
    requests:
      cpu: 1
      memory: 4Gi
username: app@example.org
```

Here you’ll see that the various properties under “spec” from the base version of cluster.yaml have been preserved in our new variant. Meanwhile, we’ve added a namespace and the additional properties for the resource limits and requests.

This strategy preserves the common spec settings in the base so that they don’t appear in more than one place. We can also do things like adjust names with namePrefix or create complex hierarchy of changes, patches via strategic merges, and addition of resources like deployment-specific secrets.
How to use kustomize

Kustomize is a tool that fits in with the idea of “configuration as code”. There is a wide range of ways that kustomize can be used successfully, and it depends on where your project or organization is in terms of automation. These configurations are compatible with many CI/CD tools for Kubernetes such as Flux CD.

The simplest way is to run kustomize with kubectl apply -k. In the example, we could create our development cluster by:

```
kubectl apply -k dev
```

If we need to feed the output to another tool, the output of kubectl kustomize has the appropriate separators to be used as a bundle. We can easily redirect the output for use in another tool:

```
kubectl kustomize dev > bundle.yaml
```

Finally, you can run kustomize as its own binary outside of kubectl. The “kustomize build” command has the same outcome as “kubectl kustomize”:

```
kustomize build dev
```

Note: you can install kustomize directly in a variety of ways (e.g. via homebrew on the Mac).

Updated: November 5, 2021