Declarative Kubernetes Manifests

Summary

Kubernetes is widely known for its support for imperative and declarative management techniques. The **imperative** approach enables the management of resources using kubectl commands directly on the live cluster. For example, all the commands you have practiced so far (e.g. kubectl create deploy [...]) are using the imperative approach. This technique is best suited for development stages only, as it presents a low entry-level bar to interact with the cluster.
On the other side, the **declarative** approach uses manifests stored locally to create and manage Kubertenest objects. This approach is recommended for production releases, as we can version control the state of the deployed resources. However, this technique presents a high learning curve, as an in-depth understanding of the YAML manifest structure is required. Additionally, using YAML manifests unlocks the possibility of configuring more advanced options, such as volume mounts, readiness and liveness probes, etc.

YAML Manifest structure

A YAML manifest consists of 4 obligatory sections:

- **apiversion** - API version used to create a Kubernetes object
- **kind** - object type to be created or configured
- **metadata** - stores data that makes the object identifiable, such as its name, namespace, and labels
- **spec** - defines the desired configuration state of the resource

To get the YAML manifest of any resource within the cluster, use the `kubectl get` command, associated with the `-o yaml` flag, which requests the output in YAML format. Additionally, to explore all configurable parameters for a resource it is highly recommended to reference the official Kubernetes documentation.

Deployment YAML manifest

In addition to the required sections of a YAML manifest, a Deployment resource covers the configuration of ReplicaSet, RollingOut strategy, and containers. Bellow is a full manifest of a Deployment explaining each parameter:

```
## Set the API endpoint used to create the Deployment resource.
apiVersion: apps/v1
## Define the type of the resource.
kind: Deployment
## Set the parameters that make the object identifiable, such as its
name, namespace, and labels.
metadata:
  annotations:
  labels:
    app: go-helloworld
  name: go-helloworld
  namespace: default
## Define the desired configuration for the Deployment resource.
spec:
  ## Set the number of replicas.
  ## This will create a ReplicaSet that will manage 3 pods of the Go
hello-world application.
  replicas: 3
  ## Identify the pods managed by this Deployment using the following
selectors.
  ## In this case, all pods with the label `go-helloworld`.
  selector:
    matchLabels:
      app: go-helloworld
  ## Set the RollingOut strategy for the Deployment.
  ## For example, roll out only 25% of the new pods at a time.
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
```

```
  ## Set the configuration for the pods.
  template:
    ## Define the identifiable metadata for the pods.
    ## For example, all pods should have the label `go-helloworld`
    metadata:
      labels:
        app: go-helloworld
    ## Define the desired state of the pod configuration.
    spec:
      containers:
        ## Set the image to be executed inside the container and image
pull policy
        ## In this case, run the `go-helloworld` application in version
v2.0.0 and
        ## only pull the image if it's not available on the current
host.
      - image: pixelpotato/go-helloworld:v2.0.0
        imagePullPolicy: IfNotPresent
        name: go-helloworld
        ## Expose the port the container is listening on.
        ## For example, exposing the application port 6112 via TCP.
        ports:
        - containerPort: 6112
          protocol: TCP
        ## Define the rules for the liveness probes.
        ## For example, verify the application on the main route `/`,
        ## on application port 6112. If the application is not
responsive, then the pod will be restarted automatically.
        livenessProbe:
           httpGet:
             path: /
             port: 6112
        ## Define the rules for the readiness probes.
        ## For example, verify the application on the main route `/`,
        ## on application port 6112. If the application is responsive,
then traffic will be sent to this pod.
```

```
       readinessProbe:
         httpGet:
           path: /
           port: 6112
       ## Set the resource requests and limits for an application.
       resources:
       ## The resource requests guarantees that the desired amount
       ## CPU and memory is allocated for a pod. In this example,
       ## the pod will be allocated with 64 Mebibytes and 250
miliCPUs.
         requests:
           memory: "64Mi"
           cpu: "250m"
       ## The resource limits ensure that the application is not
consuming
       ## more than the specified CPU and memory values. In this
example,
       ## the pod will not surpass 128 Mebibytes and 500 miliCPUs.
         limits:
           memory: "128Mi"
           cpu: "500m"
```

Service YAML manifest

In addition to the required sections of a YAML manifest, a Service resource covers the configuration of service type and ports the service should configure. Bellow is a full manifest of a Service explaining each parameter:

```
## Set the API endpoint used to create the Service resource.
apiVersion: v1
## Define the type of the resource.
kind: Service
## Set the parameters that make the object identifiable, such as its
name, namespace, and labels.
metadata:
  labels:
    app: go-helloworld
```

```
  name: go-helloworld
  namespace: default
## Define the desired configuration for the Service resource.
spec:
  ## Define the ports that the service should serve on.
  ## For example, the service is exposed on port 8111, and
  ## directs the traffic to the pods on port 6112, using TCP.
  ports:
  - port: 8111
    protocol: TCP
    targetPort: 6112
  ## Identify the pods managed by this Service using the following
selectors.
  ## In this case, all pods with the label `go-helloworld`.
  selector:
    app: go-helloworld
  ## Define the Service type, here set to ClusterIP.
  type: ClusterIP
```

Useful command

Kubernetes YAML manifests can be created using the `kubectl apply` command, with the following syntax:

```
# create a resource defined in the YAML manifests with the name
manifest.yaml
kubectl apply -f manifest.yaml
```

To delete a resource using a YAML manifest, the `kubectl delete` command, with the following syntax:

```
# delete a resource defined in the YAML manifests with the name
manifest.yaml
kubectl delete -f manifest.yaml
```

Kubernetes documentation is the best place to explore the available parameters for YAML manifests. However, a support YAML template can be constructed using kubectl commands. This is possible by using the `--dry-run=client` and `-o yaml` flags which instructs that the command should be evaluated on the client-side only and output the result in YAML format.

```
# get YAML template for a resource
kubectl create RESOURCE [REQUIRED FLAGS] --dry-run=client -o yaml
```

For example, to get the template for a Deployment resource, we need to use the create command, pass the required parameters, and associated with the `--dry-run` and `-o yaml` flags. This outputs the base template, which can be used further for more advanced configuration.

```
# get the base YAML templated for a demo Deployment running a nxing application
 kubectl create deploy demo --image=nginx --dry-run=client -o yaml
```

Declarative Kubernetes Manifests Walkthrough

Summary

This demo showcases how a Namespace and Deployment resource can be deployed using YAML manifests.

New terms

- **Imperative configuration** - resource management technique, that operates and interacts directly with the live objects within the cluster.
- **Declarative configuration** - resource management technique, that operates and manages resources using YAML manifests stored locally.

Further reading

Explore more details about different management techniques and advanced configuration for Deployment resources:

- Managing Kubernetes Objects Using Imperative Commands
- Declarative Management of Kubernetes Objects Using Configuration Files
- Configure Liveness, Readiness Probes for a Deployment
- Managing Resources for Containers