Imperative/Declarative and a Few `kubectl` tricks

Sebastien Goasguen Follow
Feb 1, 2018 · 6 min read

Kubernetes object management falls under the often criptic imperative vs declarative modes. It is a misunderstood aspect of k8s for beginners and it deserves a little more clarity. In this post I want to highlight what this means and show you how to use `kubectl` to migrate from one mode to the other.

PS: There is nothing magic in this post and it is mostly a summary of what you can find in the upstream documentation.

TL;DR just use infrastucture as code :) and just because my kids play Pokemon cards here is a little Pikachu:

Imperative

The first mode for managing objects is to use the CLI and issue what we call *imperative commands*, what this means is that objects are created and managed/modified using the CLI. All operations are done on live objects.

For example to create a namespace, a quota, a deployment and a service we can use the following four CLI commands:

```
kubectl create ns ghost
kubectl create quota blog --hard=pods=1 -n ghost
kubectl run ghost --image=ghost -n ghost
kubectl expose deployments ghost --port 2368 --type LoadBalancer -n
ghost
```

To modify any of the objects you can use the `kubectl edit` command or use any of the convenience wrappers. For example to scale the deployment do:

```
kubectl scale deployment ghost --replicas 2 -n ghost
```

If you already know some of the Kubernetes objects, you can use the `kubectl create` command which has a few handy wrappers. You can easily create a *configmap*, a *serviceaccount*, a *role* and a few other objects. For example to create a service do:

```
kubectl create service clusterip foobar --tcp=80:80
```

To create a single Pod, the `kubectl run` command as an option `--generator` which can be very handy.

```
kubectl run --generator=run-pod/v1 foobar --image=nginx
```

However you will see very quickly that these CLI wrappers are very limiting. The full schema is not configurable. For example with `kubectl run` you cannot create a Pod with multiple containers and you cannot create volumes. Only a few parameters are configurable via the CLI, for example container resource requests and service account name:

```
kubectl run --generator=run-pod/v1 foobar --image=nginx —
serviceaccount=foobar --requests=cpu=100m,memory=256Mi
```

This is also something that you can see with the `docker run` command. This usually

leads to bloating of the CLI and complex CLI commands to create objects. An anti-pattern then emerges, with users writing shell scripts to simplify the CLI use.

While the CLI is very powerful and is terrific to get on-board Kubernetes, the clear disadvantage is the lack of review process for action done on the cluster and the lack of *source of truth* for what should be running. The logical next step is to start using the full manifests for all objects existing in the cluster.

Imperative with configuration files

If you want to *migrate* from having managed your objects from the CLI, you can *export*_the manifests using a little known option of `kubectl get` , namely the `--export` option. It removes the status field and things like timestamp. You can do this by hand by `--export` saves you time. Try it:

```
$ kubectl get deployments ghost --export -n ghost -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 annotations:
 deployment.kubernetes.io/revision: “1”
 creationTimestamp: null
 generation: 1
 labels:
 run: ghost
 name: ghost
 selfLink: /apis/extensions/v1beta1/namespaces/ghost/deployments
/ghost
spec:
…
```

You can then save the manifest in a file, modify it and replace the live objects with a workflow of:

```
$ kubectl get deployments ghost --export -n ghost -o yaml >
ghost.yaml
$ vi ghost.yaml
$ kubectl replace -f ghost.yaml
```

If you are creating an object from scratch you can open an editor and start writing your manifest, or you can use one of the generators and leverage the `--dry-run` command. Super handy ! For example to create a manifest for a ClusterIP service do the following:

```
kubectl create service clusterip foobar --tcp=80:80 -o json --dry-run
{
 “kind”: “Service”,
 “apiVersion”: “v1”,
 “metadata”: {
 “name”: “foobar”,
 “creationTimestamp”: null,
 “labels”: {
 “app”: “foobar”
 }
 },
 “spec”: {
 “ports”: [
 {
 “name”: “80–80”,
 “protocol”: “TCP”,
 “port”: 80,
 “targetPort”: 80
 }
 ],
 “selector”: {
 “app”: “foobar”
 },
 “type”: “ClusterIP”
 },
 “status”: {
 “loadBalancer”: {}
 }
}
```

The `--dry-run` option also works with the `kubectl run` command. That way you can easily get the basic skeleton of a Deployment, Job or Pod.

Then once you have written all your manifests and that you are starting complaining about the **“face full of YAML problem”** you can start creating your objects. In this still imperative mode you need to tell Kubernetes what to do with the object `create` or `delete` or `replace` .

For Deployments use the `--record` option, it will prove semi-handy later when you

check your Deployment history. Using the `--record` will add an annotation to the object which will be used as *CHANGE-CAUSE* of a revision. Note that you can add the annotation later as well using the `kubectl annotate` command.

To look out for a future where you may start using a full declarative mode, use the `--save-config` option when you `kubectl create` . This will store the object configuration as an annotation.

In this mode, we now have access to the full schema of every object. It is great but at the same time requires the users to learn the API fully. We can also use version control for change management on the manifests and hence have a source of truth and an audit trail.

The biggest drawback is that if you change a manifest, you need to replace the entire live object using `kubectl replace` . If you (or something) updated the configuration out of band of that process, say using the CLI, then you will loose the state of the live objects.

Case in point:

```
kubectl scale deployment ghost -n ghost — replicas 2
kubectl get pods -n ghost
NAME READY STATUS RESTARTS AGE
ghost-8449997474–65699 1/1 Running 0 3m
ghost-8449997474-t856r 1/1 Running 1 20h
vi ghost.yaml #change the image
kubectl replace -f ghost.yaml -n ghost
kubectl get pods -n ghost
NAME READY STATUS RESTARTS AGE
ghost-5459464f7b-xjs74 0/1 ContainerCreating 0 2s
ghost-8449997474–65699 0/1 Terminating 0 5m
ghost-8449997474-t856r 0/1 Terminating 1 20h
```

The moral of the story here is: never mix mode of object management. If you start using configuration files and `kubectl replace` , never modify live objects using the CLI directly.

This also implies that if you have some automation, say a Pod auto-scaler, which modifies live object then this mode of operation will be very challenging as modification will need to be reflected in the manifests in order to not be lost at the next update….Ouch !!

Declarative

To solve the problem of keeping track of changes to live objects by the system itself (multiple writers problem), one should use a fully declarative mode. In this mode, the creation, deletion and modification of objects is done via a single command

```
kubectl apply -f <object>.<yaml,json>
```

With the `apply` command the configuration will be saved in an annotation (`*kubectl.kubernetes.io/last-applied-configuration*`) and used during three way merges of changes. Kubernetes will check the state of the live object, the configuration stored in the annotation and the manifest being provided. It will then perform some advanced patching to modify only the fields that need to be modified.

There is an advanced discussion about how `apply` calculates differences and merges changes in the documentation.

Kubernetes Declarative Programming Imperative Programming Kubectl Infrastructure As Code

Note that to delete an object it still recommended to be very explicit and use the `kubectl delete` imperative command on a specify manifest. Note to self :) do not `rm` the manifest file, use `*kubectl delete -f <object>.<yaml,json>*` :)

Get started with the CLI convenience wrappers/generators like `kubectl run` and `kubectl expose` and a few `kubectl create`, then export your object configuration as manifest files using the `--export`, or re-generate them using the `--dry-run` command. You will then have all your manifests available and you can store them in version control.

Then setup you jenkins jobs :) and run `kubectl create`. But you will most likely run into issues if you let people update the objects live from the CLI, so move to a `kubectl apply` mode.

The final straw is that you will complain about authoring manifest file and maintaining them and before you start writing some tools to facilitate that please join the App-Def

working group of attend SIG-apps.