**Hugues Alary** 

2020/05/15

In this post I walk through the current commonly accepted way of organizing Kubernetes manifest files and propose a new approach that brings consistency, modularity and adaptability to our Kubernetes application manifests.

If you don't feel like reading a lenghty blog post and wish to get to the meat right away, jump straight to the conclusion and my list of recommendations.

## The current way

A common way of organizing kubernetes manifests files in a folder often looks like the following:

At first glance and for a small application this might seem like a fine structure. However, as the application grows in size and complexity, this organization quickly breaks down.

A more mature application can easily be composed of a half dozen Deployments, a few ConfigMaps, HorizontalPodAutoscalers, many Services, Ingresses, CronJobs, StatefulSets, PersistentVolumeClaims, etc.

As the number of kubernetes objects grows, so does the number of files, with each files often time grouping multiple object declarations.

For example, here's the <u>nginx-controller helm chart</u>:

```
— Chart.yaml
— OWNERS
— README.md
— ci
— // Folder left out for brievity
— templates
— NOTES.txt
— _helpers.tpl
— addheaders-configmap.yaml
— admission-webhooks
— job-patch
— clusterrole.yaml
— clusterrolebinding.yaml
— job-createSecret.yaml
— job-patchWebhook.yaml
```

```
psp.yaml
             role.yaml
             rolebinding.yaml

    serviceaccount.yaml

       - validating-webhook.yaml
  clusterrole.yaml

    clusterrolebinding.yaml

    controller-configmap.yaml

    controller-daemonset.yaml

    controller-deployment.yaml

  controller-hpa.yaml
  controller-metrics-service.yaml

    controller-poddisruptionbudget.yaml

    controller-prometheusrules.yaml

  controller-psp.yaml
  controller-role.yaml

    controller-rolebinding.vaml

  controller-service.yaml

    controller-serviceaccount.yaml

    controller-servicemonitor.yaml

    controller-webhook-service.yaml

    default-backend-deployment.yaml

    default-backend-poddisruptionbudget.yaml

    default-backend-psp.yaml

  default-backend-role.yaml

    default-backend-rolebinding.yaml

  default-backend-service.yaml

    default-backend-serviceaccount.yaml

  proxyheaders-configmap.yaml
   tcp-configmap.yaml
  udp-configmap.yaml
values.yaml
```

It can raplidly become hard to make sense of the overall architecture of the application: which Deployment is this Service associated with? What about this PersistentVolumeClaim, where is it being used? Does this Deployment have a HorizontalPodAutoscaler associated with it?

Often time the answer is conveyed by the filename itself. For example, a Deployment might be named nging-deployment.yaml and its associated Service might be called nginx-service.yaml. However, with the number of files growing in the directory, it can get particularly hard to spot a file, especially when said file, actually, does not exist.

Further, the nomenclature between repositories is often times inconsistent: while, for example, some developers use the suffix -service to describe Service maniftest files, others use -svc, and some prefer using a prefix.

## A better way

I have been organizing my Kubernetes files in a way that I believe to be superior to the current state of affairs and my hope is that by presenting it here it might see some adoption.

Here's the basic *directory* structure:

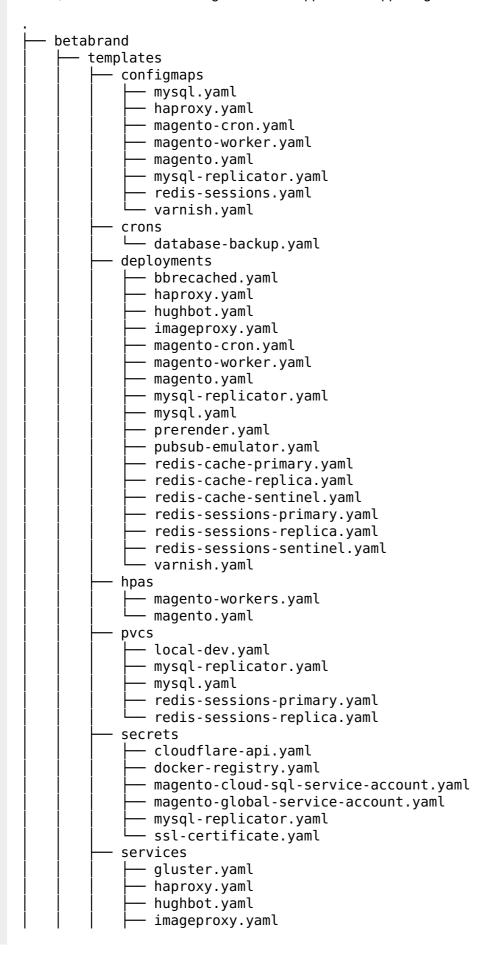
```
my-kubernetes-app
configmaps
crons
deployments
hpas
pdbs
podpriorities
pvcs
services
statefulsets
```

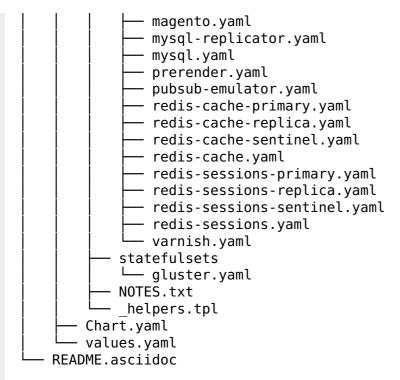
Each line in the figure above corresponds to a directory, with each folder containing a very specific kind of Kubernetes object.

With this structure, it becomes much simpler to reason about the design of an application made of many manifest files.

Let's illustrate this with a real world example.

Below, the helm chart of the large *monolithic* application supporting Betabrand.com:





In the figure above, the overall architecture of the application is distinguishable at a glance as well as the software stack: mysql, haproxy, magento, bbrecached (an internal software), hughbot (another internal tool), imageproxy, prerender, google pub-sub emulator, redis, varnish and gluster.

It is made obvious that to the 18 Deployments correspond 18 Services, 8 ConfigMaps are easily matched to their 8 Deployments, a lone StatefulSet does not have any corresponding configuration, etc.

There are 2 HorizontalPodAutoscalers; can you guess which Deployments they manage autoscaling for?

Also, omitted from the previous figure but actually present in the real Betabrand repository: a cluster-config directory describing Kubernetes objects that I consider aren't part of the application itself as they are concerned about the cluster configuration itself. The same structure applies here too.

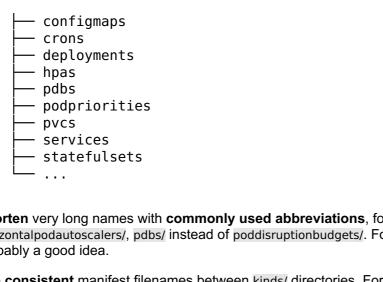
```
cluster-config
     clusterrolebindings
       external-dns.yaml
       tiller.yaml
     clusterroles
       — external-dns.yaml
     deployments
       external-dns.yaml
     podpriorities
     └─ high-priority.yaml
     serviceaccounts
       — external-dns.yaml
     storageclasses
        persistent-fast.yaml
       persistent.yaml
     README.asciidoc
```

## Conclusion

So here it is, my recommendations when it comes to organizing your Kubernetes manifests:

• **Group** manifest files in **directories named after the** *Kind* **of object**: deployments, configmaps, services, etc. Note the directory name lower cased and pluralized.

```
.
└─ my-kubernetes-app
```



- Shorten very long names with commonly used abbreviations, for example, hpas/ instead of horizontalpodautoscalers/, pdbs/ instead of poddisruptionbudgets/. Following the kubectl nomenclature is probably a good idea.
- Use consistent manifest filenames between kinds/ directories. For example, if your app is named funnygifs-slackbot:
  - o the Deployment should be named funnygifs-slackbot.yaml
  - the Service funnygifs-slackbot.yaml
  - the ConfigMap funnygifs-slackbot.yaml
  - o the HPA funnygifs-slackbot.yaml
  - the PVC funnygifs-slackbot.yaml
  - o etc.
- Avoid stutter: do not call a deployment manifest funnygifs-slackbot-deploy.yaml or a service funnygifslackbot-service.yaml, etc.
- Use resource names consistent with the manifest filename:

deployments/funnygifs-slackbot.yaml apiVersion: apps/v1 kind: Deployment metadata: name: funnygifs-slackbot (1) labels: app: funnygifs-slackbot

1. name: matches the filename

services/funnygifs-slackbot.yaml

apiVersion: v1 kind: Service metadata:

name: funnygifs-slackbot (1)

1. name: matches the filename

configmaps/funnygifs-slackbot.yaml

apiVersion: v1 kind: ConfigMap metadata:

name: funnygifs-slackbot (1)

- 1. name: matches the filename
  - etc.

© 2019-2022 Hugues Alary