Kubernetes ❤️ Python

Kubernetes Workshop EuroPython 2023

Oz Tiram

Preface

This document was written for the Kubernetes workshop in EuroPython 2023, Prague.

Credits

This document was written using Sphinx using the blue-calm theme. The blue-calm theme is based on the Sphinx-Business-Theme project.
The cover photo is taken from Unsplash by @davidclode It shows the skin of a water python which has just shed its old skin, revealing the beautiful new iridescent skin underneath. The Australian Aboriginal people have myths and legends about a rainbow serpent, and perhaps a sight like this inspired them. Parts of this workshop are based on the The little book of operators and article previously published in the German developer magazine etwickler.de as well as my blog post in Ensligh.

Table of Contents

Introduction

What is Kubernetes?

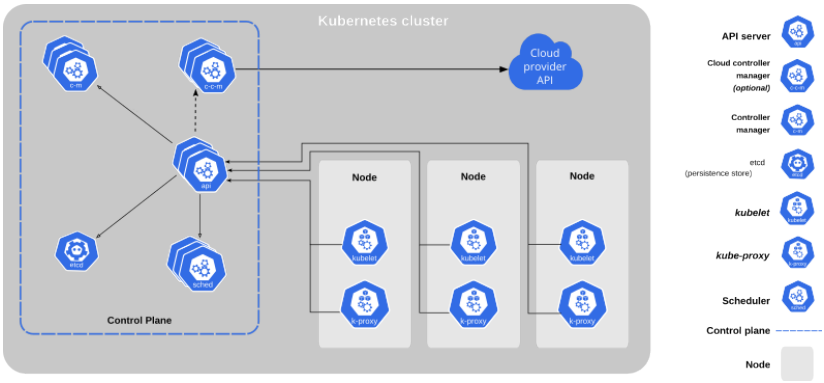
Originally developed internally by google as a project known Borg, Kubernetes is an open source platform, to schedule workloads on a cluster of computers. The most basic unit of work that kubernetes understands is a Pod, which is originally, one or more docker containers with dedicated amount of compute resources. These originally include CPU, RAM, network and disks. This unit of work is the source of viewing Kubernetes as a Container Scheduler. However, since the inception and birth of Kubernetes, work units have become more than just docker containers. In the meanwhile Kubernetes, can schedule a lot of things. For example LXC containers, Full blown virtual machines, lightweight virtual machines, and even FreeBSD Jails.

As such, we should view Kubernetes as a Workload Scheduler or as a distributed kernel, where we ask the system for compute resources to execute different tasks. Kubernetes then

tries to fulfill our request by allocating compute resources and running our tasks.

How is it built?

Kubernetes is a distributed system built from worker and master nodes. Each system, is a node in the cluster, and can be either a worker or a master. The master node is responsible for the orchestration of the cluster, and the worker nodes are responsible for the execution of the workloads.



How does it work?

Kubernetes is a distributed system, and as such, it is built from multiple components, that work together to provide the functionality of the system. The main components are:

- **etcd** - A distributed key value store, that is used to store the state of the cluster.
- **kube-apiserver** - The API server is the main entry point to the cluster. It is the only component that is exposed to the outside world. It is the component that is responsible for the communication with the etcd cluster. It is also the component that is responsible for the authentication and authorization of the users.
- **kube-controller-manager** - The controller manager is the component that is responsible for the reconciliation loop of the cluster. It is the component that is responsible for the creation of the resources in the cluster.
- **kube-scheduler** - The scheduler is the component that is responsible for the scheduling of the workloads on the cluster. It is the component that decides on which node the workload should run.
- **kubelet** - The kubelet is the component that is responsible for the execution of the workloads on the node. It is the

component that is responsible for the creation of the containers on the node.

- **kube-proxy** - The proxy is the component that is responsible for the networking of the cluster. It is the component that is responsible for the creation of the network rules on the node.

This distributed kernel is composed of many controllers whose responsibility is fulfilling a desired state and react to changes in the actual state of what ever resources they control. If you already work with Kubernetes, you are probably familiar with the built-in controllers `Deployment` and `StatefulSet`

How do I use it?

To change the state of the cluster or see the state of the cluster you need to modify or query the data storage, namely etcd. It is impractical to it directly. Hence, the *kube-api-server* is the main entry point to the cluster. All components of the cluster comunicate with each other via a RESTful API. Administrators and users of the clustere use a command line tool *kubectl* to translate message to or get infomation from the cluster via the command line. Software developers can use client libraries which translate methods to JSON payloads or which translate JSON payloads to objects.

Why is it useful?

Kubernetes provides a consitent way to interact with compute resources - on bare metal or public clouds. It provides fault tolerency with a relative low entry barrier for single developer or teams whether large or small. It has a large set of API which abstract Network, CPU, Disk and RAM requests in a declarative way. I like to think about Kubernetes as an ORM for Public Clouds and Bare Metal. It allows one to request compute resources for a workload - or an app if you will - and Kubernetes will try and eventually provision the resources and run the workload. This is extremely useful, and allows teams to describe the complete life cycle of an application's

infrastracture in a delerative way. The idea of Infrastructure as code isn't new, nor declerative programming.

Declarative

As mentioned above the idea of declerative IoC isn't knew. However, YAML manifest are definitely more declarative than let's say for example Ansible Playbooks or Roles, or even Salt's states.

For a longer discussion about Decl$arative vs. Imperative see the excellent blog post: Is it Imperative to be Declarative?

Eventually Consistent

In a very simplyfied view, users of Kubernetes use the command line or client library to declare a desired state of the system, for example, run 3 replicas of my Django app. Kubernetes takes this request and passes it to a series of controllers which take actions to fulfill this desired state. After the desired state is reached, the controllers keep watching the state and take actions to re-adjust. For example if one replica of the application has crashed, a new replica will be started.

Sometimes, certain conditions can not be met. Hence, Kuberenets will leave things as is for a while, but then try again until conditions are met. This can happen, for example, when a user requests a work load then which requires more CPU and RAM available in the cluster. In such case, if configured, a controller capable of starting compute resources (e.g. cloud controller), will start a desired VM configuration. Once up and running, the application controller will start the workload on this machine.

Deploying Kubernetes

Depite being a distributed system by design, it is not hard to deploy Kubernetes.

Cloud Hosted

Almost all public clouds offer a managed Kubernetes as a service. It requires zero knowledge to administer. You can launch a cluster using terraform, command line utils (e.g gcp cli, aws cli, or digital ocean and others.) While easy to start with these Kubernetes services might impose their choice of configuration, and also cost more. For example most managed Kubernetes services impose a control plane with 3 nodes, which might be expensive for small team.

Self Managed

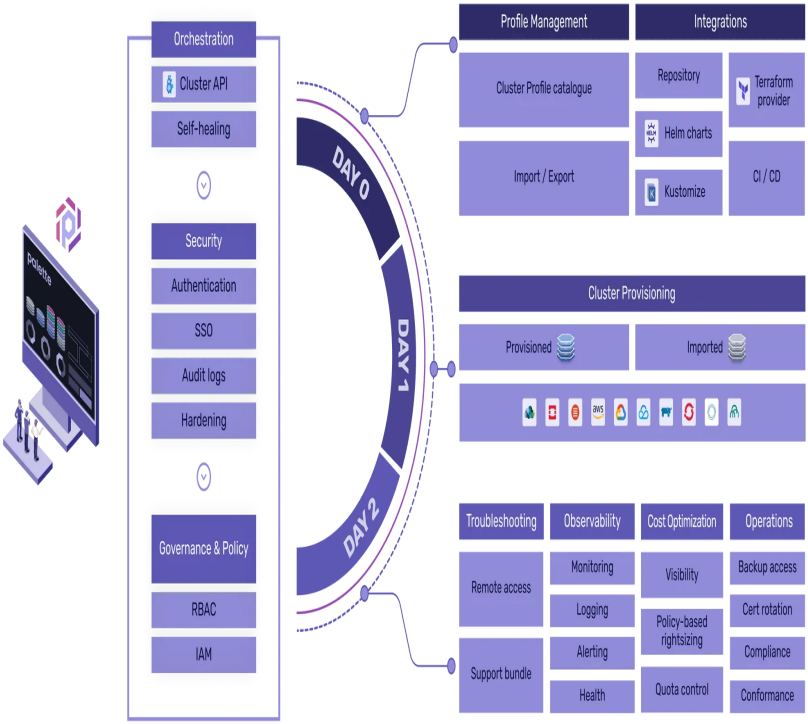
One can also choose to install kuberentes on his or her own. That is a more involved process with varying degree of difficulty. Whether using bare metal machine or VM, whether one has to configure the network switches or cloud VPC will determine how hard is it to complete all the pre-required steps to install Kubernets. Once all components are in place, the official Kubernets installer, kubeadm can be used to provision the Kuberentes control plane and worker nodes.

Choosing this path requires also thinking about day-2 operations. That means, not only installing Kubernetes, but also updating and patching the OS, Kubernetes componentes, and upgrading the cluster when a software version turns outdated.

Palette PaaS

Palette PaaS offers small and large team a cloud agnostic platform to manage the complete life cycle of a single cluster or multiple Kubernetes clusters. It manages Kubernetes cluster across multiple Cloud Provider and Bare metal machines as they were cloud. It allows team to declare the complete cluster and the application using manifests similar

to Kubernetes manifest, and takes away the complexity of managing day-2 operations of Kuberentes.



How many Nodes?

Despite being designed as distributed system, you can install a single node Kubernetes. That means you can start small and grow as you need. By doing that you can enjoy Kubernetes on a cheap VPS or a Raspberry Pi, without the extra costs for a complete VPC or expensive networking and server hardare.

For this you can use the excelent microk8s or k3s distributions of Kubernetes.

Local development

For local development you can use kind or minikube. Both allow you to quickly launch a local Kuberentes on your local machine.. They are great for testing and development of Kubernetes manifests and applications. kind is lighter since it runs the Kubernetes components as single docker container. *minikube* runs the Kunbernetes component using on of multiple backends. You can use one of many backends (Docker or VirtualBox, or KVM, or Hyper-V, or VMware Fusion, or VMware Workstation) depending on your host OS) to create a cluster with 1 or more nodes locally. In this workshop I will use *minikube* using the *kvm* backend on Linux.

Setting minikube up

To practice using Kubernetes and extending it you should have access to a Kubernetes cluster. Ideally, one where you have admin rights, and you are not afraid of wrecking havoc.

minikube offers a way to quickly set up a Kubernetes cluster on your developer machine.

Follow the installation guide to get minikube installed. For starting a cluster on a Windows machines you need either Docker or VirtualBox installed to.

I will be using Gentoo and the KVM backend for the demo, however all commands should work on your OS too.

Starting a kubernetes cluster

In your terminal shell do:

```
$ minikube config set memory 2848
$ minikube config set cpus 2
$ minikube config set vm-driver kvm2
```

You should see something similar:

```
 $ minikube start -p europython
😄  [europython] minikube v1.25.2 on Gentoo 2.8
✨  Using the kvm2 driver based on user configuration
👍  Starting control plane node europython in cluster europython
🔥  Creating kvm2 VM (CPUs=2, Memory=2848MB, Disk=20000MB) ...
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
  ▪ kubelet.housekeeping-interval=5m
  ▪ Generating certificates and keys ...
  ▪ Booting up control plane ...
  ▪ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
  ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "europython" cluster and "default"
namespace by default
```

You can add more nodes to the cluster with:

```
$ minikube node add -p europython
😄  Adding node m02 to cluster europython
❗  Cluster was created without any CNI, adding a node to it might cause broken networking.
👍  Starting worker node europython-m02 in cluster europython
🔥  Creating kvm2 VM (CPUs=2, Memory=2848MB, Disk=20000MB) ...
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
🔎  Verifying Kubernetes components...
🏄  Successfully added m02 to europython!
```

Assuming you already installed *kubectl*, you can now view all the nodes in your cluster:

```
$ kubectl get nodes
NAME             STATUS   ROLES                  AGE     VERSION
europython       Ready    control-plane,master   2m32s   v1.23.3
europython-m02   Ready    <none>                 99s     v1.23.3
```

✍️ Exercise

check that your minikube or cluster is ready to start working with:

```
$ kubectl get pod -A | awk '{printf "%-40s %-70s %-10s %-20s\n", $1, $2, $3, $6}'
```

✔ Solution

The output should be similar to this one:

```
$ kubectl get pod -A | awk '{printf "%-40s %-70s %-10s %-20s\n", $1, $2, $3, $6}'
NAMESPACE     NAME                                      READY   STATUS      AGE
kube-system   coredns-64897985d-xrh76                   1/1     Running     3m23s
kube-system   etcd-europython                           1/1     Running     3m37s
kube-system   kindnet-sg4zz                             1/1     Running     2m41s
kube-system   kindnet-x5m78                             1/1     Running     2m41s
kube-system   kube-apiserver-europython                 1/1     Running     3m36s
kube-system   kube-controller-manager-europython        1/1     Running     3m34s
kube-system   kube-proxy-6s5qs                          1/1     Running     2m46s
kube-system   kube-proxy-jv2tv                          1/1     Running     3m22s
kube-system   kube-scheduler-europython                 1/1     Running     3m34s
kube-system   storage-provisioner                       1/1     Running     3m33s
```

Workloads

The minikube cluster we deployed already has some workloads running. They are all in the namespace *kube-system* and you can see some of the components we discussed earlier.

Containers and Pods

By default, Kuberenetes uses Containers to distribute workloads to nodes on the clusters.

I am not going to discuss Linux Containers in length here, since we have a hard time limit. However, it's usefull to know a bit about them when working with Kubernetes. As it does help solving network isssues and running applications in a secure way in Kubernetes.

The most basic work unit that Kuberenetes understands in a *Pod*. A pod can have 1 or more containers.

This is usefull for example if you have an application which is composed of multiple components that need to have shared access to a file, or work closely together (e.g. share memory).

Another scenario, is having a *initContainer* which is responsible for a short living pre-task, and another long running process.

Here is an example for a Pod with multiple containers:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-first-k8s-app
  labels:
    app: django-app
spec:
  initContainers:
  - name: migrator
    image: docker.io/oz123/my-catstore:0.0.1
    command: ["python3", "-u", "manage.py", "migrate"]
  containers:
  - image: docker.io/oz123/my-catstore:0.0.1
    name: catstor
    command: ["python3", "-u", "manage.py", "diffsettings", "&&", "python3", "-u",
"server.py"]
    ports:
    - containerPort: 8000
      protocol: TCP
  - image: nginx
    name: nginx
    ports:
    - containerPort: 80
      protocol: TCP
```

In this example we have a Pod with 3 containers. An *initContainer* which will execute database migrations before statring the Django application. If the migrations fail, the *Pod* will not start. If the migration will succeed, the *initContainer* will stop, and the Django app and Nginx will execute the servers which will then wait for incoming HTTP requests.

Deployments, StatefulSets, DaemonSets

Deployments

Deployments allow scaling up vertically and horizontally. By vertically, we mean a larger number of Pods. If your example, your application needs to process more requests per second, you can add more Pods, which will handle more requests. By horizontally, we mean a larger number of Pods accross multiple machines. Once a machine can no longer add more Pods, whether it's due to the physical limits of hardware, we can schedule more Pods on more machines. We can, also choose to distribute the load across machines before we reach the limit.

Here is a simplified application deployment based on the Pod spec shown earlier:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catstore-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: catstore
  template:
    metadata:
      labels:
        app: catstore
    spec:
      initContainers:
      - name: migrator
        image: docker.io/oz123/my-catstore:0.0.1
        command: ["python3", "-u", "manage.py", "migrate"]
      containers:
      - image: docker.io/oz123/my-catstore:0.0.1
        name: catstor
        command: ["python3", "-u", "manage.py", "diffsettings", "&&", "python3",
"-u", "server.py"]
        ports:
        - containerPort: 8000
          protocol: TCP
      - image: nginx
        name: nginx
        ports:
        - containerPort: 80
          protocol: TCP
```

DaemonSets

DaemonSets create a copy of a Pod on each node in the cluster. This is mostly used by cluster admistrator and less by application developers. Some example use case are:

- Installing device drivers (e.g. GPU drivers)
- Collection of Pod logs
- Running monitoring software (e.g. cAdvisor)

StatefulSets

StaefulSet are usefull for Pods that require persisting data. Hence, the name Stateful. These are useful for running databases. Here is an example StatefulSet running a PostgreSQL server:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres-sts
spec:
  serviceName: postgres-headless-svc
  replicas: 3
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgresql
          image: docker.io/bitnami/postgresql:latest
          ...
          ports:
            - name: postgresql
              containerPort: 5432
              protocol: TCP
          volumeMounts:
            - name: postgresql-vol
              mountPath: /var/lib/postgresql/data
  volumes:
   - name: postgres-vol
     awsElasticBlockStore:
       volumeID: "vol-09a8995f67b97593a"
       fsType: ext4
```

Note the volumes sections added. This is a very primitive way of attaching a volume to StatefulSet, which requires first provisioning a volume manualy in AWS and then attaching it by specifying *volumeID*.

As mentioned earlier, we can let Kubernetes manage our cloud infrastructure. For this will we use a CSI DaemonSet and and define a *PersistentVolumeClaim* and a *PersistentVolume*.

Pod Environment, ConfigMaps, Secrets

Until now, all workloads were lacking any configuration. It is possible to pass little snippets of configurations to Pods, using ConfigMap and or Secret.

Secrets are configuration files and Strings which are stored in base64 encryption. However, inside the container the are automatically decoded for the container to read. Also, secrets are always mounted as read only.

Note

Kubernetes secrets are not stored in *etcd* encrypted per default. There are, however, solutions to encrypt the data at rest or store secret in secure vaults. This is unfortunately, left out of this chapter.

Here is brief example of a Pod reading environment variables and files from ConfigMap, Secret and from the Pod manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-first-k8s-app
spec:
  containers:
  - image: docker.io/oz123/my-catstore:0.0.1
    name: catstore
    command: ["python3","server.py"]
    ports:
    ...
     env:
     - name: PYTHONUNBUFFERED
       value: "True"
     envFrom:
     - secretRef:
         name: postgresql-credentials
     - configMapRef:
         name: django-settings
     volumeMounts:
     - mountPath: /etc/s4cfg
       name: s4cfg
  volumes:
   - name: s4cfg
     secret:
       secretName: s4cfg
   - name: nginx-config-volume
     configMap:
       name: nginx-config
```

Volumes, PerstitentVolumes and Claims

We have implicitly indroduced *volumes* in the Pod manifest. These were *secret*, *configMap* and *awsElasticBlockStore*. Kubernetes has built-in volume provisioners as well as extensions that allow interaction with diffrent volume types, e.g. NFS, Local LVM, Ceph, EBS, Google Cloud storage and more. You can find more infomation about volumes and storage for containers in the Kubernetes storage documentation.

Services

Services are a way to expose Pods to the outside world. There are 3 types of services:

- *ClusterIP* - Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster.
- *NodePort* - Exposes the service on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
- *LoadBalancer* - Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

Here is an example of a Service manifest:

```
apiVersion: v1
kind: Service
metadata:
  name: catstore-service
spec:
  type: NodePort
  selector:
    app: catstore
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8000
    nodePort: 30000
```

In this example, we have a Service which exposes the Pods with the label *app: catstore* on port 80. The service is of type *NodePort* which means that the service will be exposed on each node in the cluster on port 30000. The service will then route the traffic to the Pods on port 8000.

Ingress

Ingress is a way to expose services to the outside world. Ingress is a collection of rules that allow inbound connections to reach the cluster services. It can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, offer name based virtual hosting, and more. Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

Here is an example of an Ingress manifest:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: catstore-ingress
spec:
  rules:
  - host: catstore.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: catstore-service
          servicePort: 80
```

All together

We now, briefly, introduced all the componentes required for running a Django application on Kubernetes.

✍️ Exercise

Clone the repository at:

https://gitlab.com/oz123/catster

It contains a simple Django based application that shows a cat image.

Build the application use the *make* command. Apply all kubernetes manifests are found in the directory *k8s*.

Extending Kuberenetes

There are three ways in which we can extend Kubernetes using Python.

The first one is adding functionallity to *kubectl* via plugins. The second one modifies the behaviour of the API server by adding *webhooks*. The third one allows us to store objects in the Kubernetes database using *CustomResourceDefinition*.

kubectl plugins

You can extend *kubectl* by writing command line programs in any languages. Python is very handy for processing data, hence it is also very handy for adding missing functionality to *kubectl*. A *kubectl* plugin is a program named *kubectl-<myplugin>* which is found in your path.

Here is a simple plugin for staring a kuberentes context aware interactive shell:

```
#!/usr/bin/env python3

import sys
import pykube

print(f"Python version: {sys.version}")

from pykube.console import main

main()
```

I can install it with:

```
install -m 755 kubectl-python /usr/local/bin
```

We'll discuss *pykube-ng* even further in the next chapter.

To run the plugin:

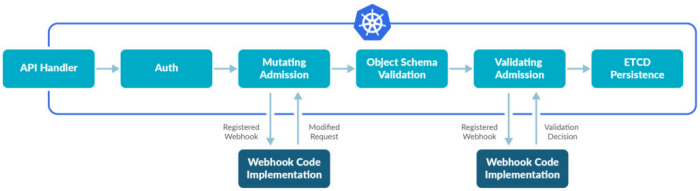
```
$ kubectl python
Python version: 3.11.4 (main, Jul  4 2023, 11:48:18) [GCC 12.2.1 20230428]
Pykube v23.6.0, loaded "/home/oznt/k8s-workshop-europython/reporting-
p2.kubeconfig" with context "reporting-p2-admin@reporting-p2".

    Example commands:
      [d.name for d in Deployment.objects(api)]              # get names of
deployments in default namespace
      list(DaemonSet.objects(api, namespace='kube-system'))  # list daemonsets in
"kube-system"
      Pod.objects(api).get_by_name('mypod').labels           # labels of pod
"mypod"

    Use Ctrl-D to exit
>>>
```

✍️ Exercise

Create a *kubctl* plugin *count-pods* which returns the count of all pods in the clusters.

Admission Webhooks

You can add behaviours to API by defining custom web hooks. These are simple web apps which are deployed inthe cluster, which validate or mutate requests before they are further processed. Unsurprisingly, the are defined by creating *ValidationAdmissionWebhook* and *MutatingAdmissionWebhook* resources.



The mutating admission hook takes action to change the API request, whereas the validating admission hook accepts or denies the request (image source).

As an example, here is a simple validating webhook:

```
import os
from bottle import Bottle, request

app = Bottle()

LABEL, KEY = os.environ['LABEL'], os.environ['ALLOW_KEY']

@app.post('/validate')
def webhook(*args, **kwargs):
    request_info = request.json
    uid = request_info["request"].get("uid")

    name = request_info["request"]["object"]["metadata"].get("name")
    kind = request_info["request"]["kind"]["kind"]
    labels = request_info["request"]["object"]["metadata"].get("labels")

    if labels and labels.get(LABEL) == KEY:
        print(f'Object {kind}/{name} contains the required
{LABEL} label. Allowing the request.')
        allowed, message = True, f"Label {LABEL} is set."
    else:
        print(f'Object {kind}/{name} doesn\'t have the required {LABEL} label.
Request rejected!')
        allowed, message = False, f"Label {LABEL} is not set!"

    return {"apiVersion": "admission.k8s.io/v1",
            "kind": "AdmissionReview",
            "response": {"allowed": allowed, "uid": uid,
                         "status": {"message": message}}}
```

✍ Exercise

Clone the repository https://gitlab.com/oz123/validating-hook-with-bottle

Deploy the Webhook to the cluster. Set up a LABEL and ALLOW key. Now, try redeploying *catster* again.

CustomResource Definitions

Very early on in the Kubernetes developers realized that allowing to extend Kubernetes is key to successful adoption. Version 1.7 of Kubernetes added the ability to define *ThirdPartyResource*, which allowed extending Kubernetes. These were later named *CustomResourceDefinition* in version 1.8 and onward.

CustomResourceDefinitions allow you to store new types in Kuberenetes.

```
apiVersion: "grafzahl.io/v1beta1"
kind: Number
metadata:
  name: total
spec:
  pods: 2
  deployments: 3
```

This can be stored in cluster with:

```
$ k apply -f numbers.yml
grafzahlnumber.grafzahl.io/total created
```

Queried with kubectl:

```
$ k get numbers.grafzahl.io
NAME    DEPLOYMENTS PODS
total   3           2
```

Here is the definition for *numbers.grafzahl.io*:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: numbers.grafzahl.io
spec:
  group: grafzahl.io
  scope: Cluster
  names:
    plural: numbers
    singular: number
    kind: Number
    shortNames:
    - gn
  versions:
  - name: v1beta1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              deployments:
                type: integer
              pods:
                type: integer
    additionalPrinterColumns:
    - name: Deployments
      type: integer
      description: Total number of deployments
      jsonPath: .spec.deployments
    - name: Pods
      type: integer
      jsonPath: .spec.pods
```

✍️ Exercise

Create your own CRD for a Restaurant. It should have the following fields:

- waiters (integeres)
- tables (integers)
- guests (integers)
- name(string)
- cuisine (one of italian or french)

Operators

By now we have, briefly, surveied the all the components required for running a web application in a Kubernetes cluster.

We also know how to extend Kubernetes in 3 different ways. In the following we will create an application that can react to changes in the cluster.

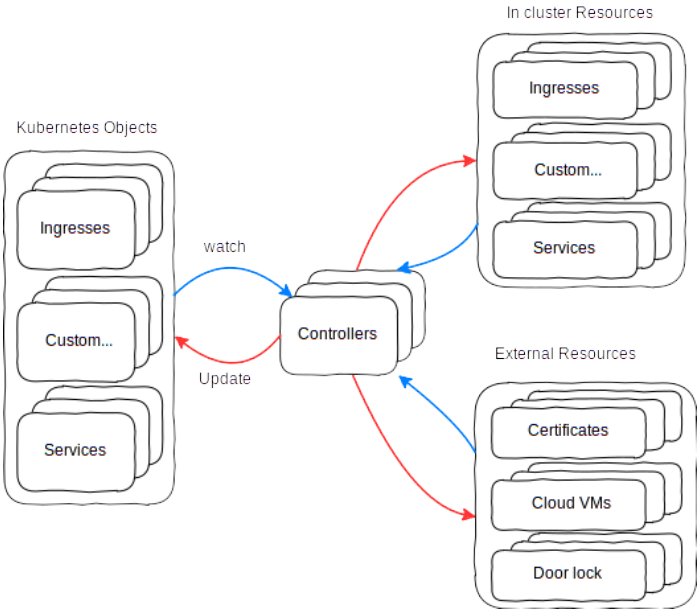
This application will connet all the components we saw before, creating a so called Operator.

An Operator is a collection of domain specific custom resources and a controller program to react to changes to the cluster or these specific resources. For example, an operator can watch certain annotations on Pods or Deployments, and manipulate objects inside or outside the cluster when these annotations are detected.

This is how CertManager or ExternalDNS work for example. Specifically, when you create an annotation on an Ingress, there is a chain of actions, which is triggered inside and outside the cluster. Along the process a certificate request is sent to LetsEncrypt and if authenticated successfully a new

secret containing a certificate is created and used to secure access to the Ingress with HTTPS.

The key message here is that: an operator can watch Kubernetes objects, built-in or custom, and act on objects, which can be external or internal to the cluster, bringing them to a desired state.



CertManager an example Operator

✍️ Exercise

Install Cert-Manager in your cluster:

```
$ k apply -f https://github.com/cert-manager/cert-manager/releases/download/
v1.12.0/cert-manager.yaml
```

Examine the CRDs it creates.

✍️ Exercise

Create a self-signed Issuer by following

https://cert-manager.io/docs/configuration/selfsigned/#bootstrapping-ca-issuers

✍️ Exercise

Modify the Ingress of *caster* so it is secured by a TLS certificate.

✍️ Exercise

Examine the *CertificateRequest* objects in the namespace *catster*:

```
$ k get certificaterequests.cert-manager.io -n catster
```

Build your own Operator in Python

Why Python?

Obviously, since kubernetes itself is written in Go, the ecosystem of operators it is also common in operators. However, you don't have to choose Go. In fact, if you are not already fluent in Go, I would advise to choose an operator framework which allows writing operators in Python. Python allows you to write code quickly and prototype your operator logic quickly. Selecting a programming language can be a form of premature optimization., hence if you like Python and your Operator isn't processing millions of request per second Python is a great choice.

About Frameworks

The blog post Frameworks don't make any sense hits a sensitive nerve I have. However, there is a delicate balance found between a large over-complex framework and completly writing everything from scratch.

Another thing about frameswork I'd like to refer too is that principles are more important than frameworks.

Never the less, an obligatory mention is Kopf.

Kopf

Kopf is a Python framework to write Kubernetes operators in Python. It is built on top of the official Kubernetes client libraries and async.io.

The architecture of Kopf is quite complex. However, it have a flasky API which will feel familiar if you have experience with microframeworks like Flask (or bottle.py or FastAPI).

✍️ Exercise

Install Kopf:

```
$ pip install kopf
```

Create a new file *operator.py* and add the following code:

```
import kopf
import kubernetes.client

@kopf.on.create('apps', 'v1', 'deployments')
def create_fn(spec, name, namespace, logger, **kwargs):
    logger.info(f'Creating deployment {name} in namespace {namespace}')
    api = kubernetes.client.AppsV1Api()
    api.create_namespaced_deployment(
        namespace=namespace,
        body=spec,
    )
```

Run the operator:

```
$ kopf run operator.py
```

Create a deployment:

```
$ k create deploy nginx --image nginx
```

Check the logs of the operator:

```
$ k logs -l app=kopf -c kopf
```

Grafzahl

Grafzahl is a small teaching operator, like Count Count from Sesame Street, it helps you count stuff in your cluster.

You already saw the *CustomResourceDefinition* of *Nubmer*. Now we will build an operator around it.

The pseudo algorithm of grafzahl is:

```
while True:
   pods = Pods.list(filter=lambda o: 'counted' not in o['labels'])
   deployments =  Deployments.list(filter=lambda o: 'counted' not in o['labels'])
   map(assign_label('counted', pods))
   map(assign_label('counted', deployents))
   update_crd_numbers(len(pods), len(deployments))
```

We will make use of the client libray pykube-ng to create this operator. After importing it, we initialize an API client:

```
import time
import pykube
DEFAULT_SCAN_DELAY=10
# automatically detect whether in cluster or load ~/.kube/config
config = pykube.KubeConfig.from_env()
```

We can now create the main to loop (listing only Deployments):

```
while True:
    for deploy in pykube.Deployment.objects(api,
                                            namespace=pykube.all,
                                            selector={'grafzahl!': 'counted'})):
        print(f'Updating deployment {deploy.namespace}/{deploy.name}..')
        deploy.labels['grafzahl'] = 'counted'
        deploy.update()

    time.sleep(DEFAULT_SCAN_DELAY)
```

pykube-ng does not know anything about our CRD *Number*. However, since Python is a dynamic language we can easily define this code even at run time:

```
class Number(pykube.objects.APIObject):
    version = "grafzahl.io/v1beta1"
    endpoint = "numbers"
    kind = "Number"
```

You see, there is no type validation and no constraints. However, nothing prevents you from adding those to the client if you want to. You can also leave it to the Kubernetes API to decide:

```
>>> total = list(Number.objects(api).filter(namespace=pykube.all))[0]
>>> total
... <Number total>
>>> total.obj['spec'] = {'deployments': 3, 'pods': 10, 'no such': 'key'}
>>> total.update()
>>> Number.objects(api).get(name="total").obj
{'apiVersion': 'grafzahl.io/v1beta1',
'kind': 'Number',
...
'operation': 'Update'...},
{'apiVersion': 'grafzahl.io/v1beta1',
'name': 'total', ...
'spec': {'deployments': 3, 'pods': 10}}
```

✍️ Exercise

Apply the Numbers CRD manifest to your cluster. Create a Number objects called *total*.

✍️ Exercise

Use the pykube-ng shell to delete *total* and recreate with Python.

✍️ Exercise

Complete the operator code above:

1. Add listing and labeling of Pod objects.

2. Update *total* count of Pod and Deployment objects.

✔ grafzahl.py

```
class Number(pykube.objects.APIObject):
   version = "grafzahl.io/v1beta1"
   endpoint = "numbers"
   kind = "Number"

config = pykube.KubeConfig.from_env()
api = pykube.HTTPClient(config)

def _init(api):
    numbers =  {"deployments": 0, "pods": 0}
    total = Number(api, obj=dict(metadata={"name":"total"},
                    kind="Number",apiVersion="grafzahl.io/
v1beta1",spec=numbers))
    total.update()

total = _init(api)

while True:
    for dx, deploy in enumerate(Deployment.objects(api,
                                                   namespace=pykube.all,
                                                   selector=SELECTOR),
                                start=1):
        print(f'Updating deployment {pod.namespace}/{pod.name}..')
        deploy.labels['grafzahl'] = 'counted'
        deploy.update()
    for px, pod in enumerate(Pod.objects(api,
                                         namespace=pykube.all,
                                         selector=SELECTOR),
                                         start=1):
        print(f'Updating pod {pod.namespace}/{pod.name}..')
        pod.labels['grafzahl'] = 'counted'
        pod.update()
    total.obj["spec"] = {"deployments": dx, "pods": px}
    total.update()
    time.sleep(DEFAULT_SCAN_DELAY)
```

Deploying Grafzahl

Obviously, for grafzahl to be useful, one has to deploy the operator to a working kubernetes cluster.

✍ Exercise

Build your container image for grafzahl.py

✍ Exercise

Create a Namespace Deployment for grafzahl.py

Make sure your operator runs.

✍ Exercise

Give grafzahl.py the correct permissions to read and update the correct objects.

Summary

By now, you have deployed a Python web application to your cluster, written kubectl command line exention, added a validating webhook and created an Operator in Python. There is still a lot of uncovered ground using Python and Kubernetes. For example, you can use https://cdk8s.io/ to generate Kubernetes manifests from Python code. We didn't cover testing operators, or highly available operators. We also didn't cover how to clean up after the operator ends its life. I hope you feel that you already know enough to find your own answers. In other words, you have enough knowledge to be dangerous. I hope you will find joy in extending Kubernetes to automate all sort of boring or interesting tasks! Finally, I hope you now no longer see Kubernetes as just a container orchestrator, but as a platform for automating all sort of interesting (or boring) tasks.

Appendix A - Setup kubectl alias

You can define aliases to common commands in your shell. It's easy create them. However, care needs to be taken, that aliases still work with tab completion.

Here is how I set my `.bashrc` for working with `k` and keeping bash completion.

First, create a bash completion file with:

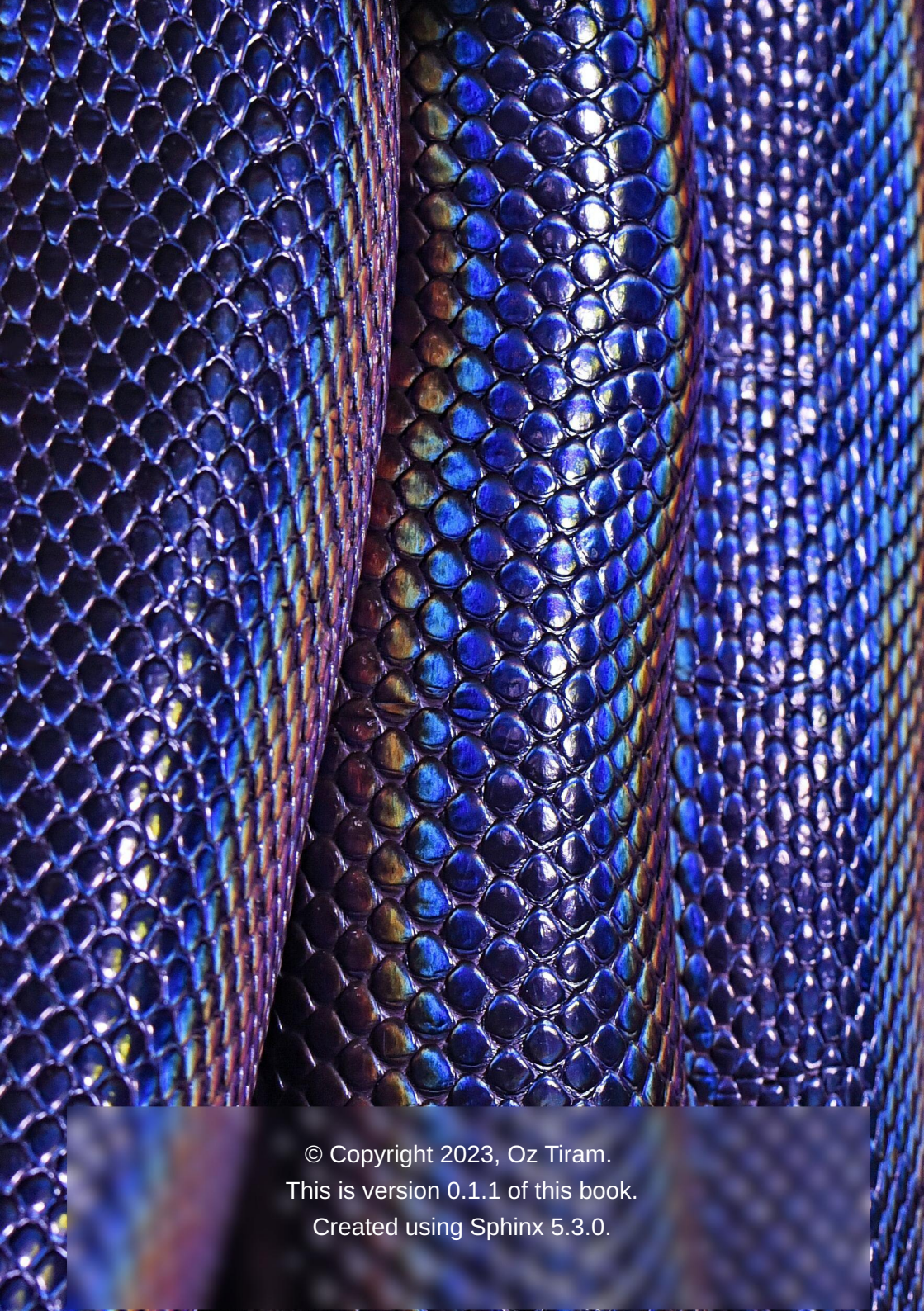
```
$ kubectl completion bash > ~/.k8s-completion.sh
```

Then, add the following snippet to your `.bashrc`:

```
if [ -r ~/.k8s.sh ]; then
    source ~/.k8s.sh
    alias k=kubectl
    complete -F __start_kubectl k
fi
```

To immidietly activate this functionality do:

```
$ source ~/.bashrc
```



This is version 0.1.1 of this book.
Created using Sphinx 5.3.0.