Nuvolaris Trainings

Developing Kubernetes Operators in Python

https://www.nuvolaris.io

Agenda

- Introducing Kubernetes Operators
- Defining CRDs and instances
- Using Kustomize for deployment
- Managing resources with the operator
- Packaging

Kubernetes Operators

Kubernetes Controllers

- Deployment, DaemonSet, StatefulSet

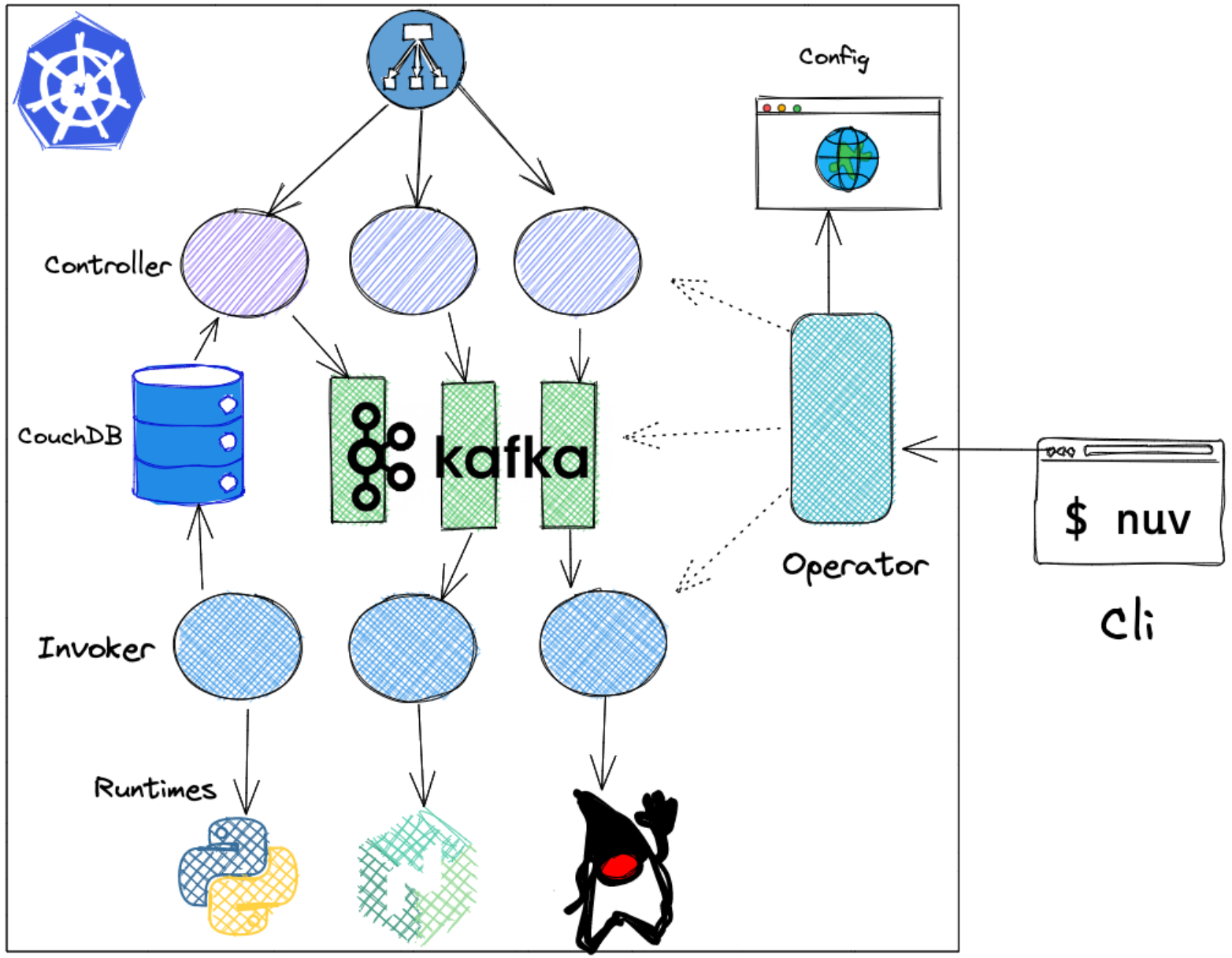
```
[~]$ kubectl get deploy
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           21m
[~]$ kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-66b6c48dd5-4dpl2   1/1     Running   1          21m
nginx-deployment-66b6c48dd5-5c4q6   1/1     Running   1          21m
nginx-deployment-66b6c48dd5-xs8nd   1/1     Running   1          21m
```

What they do?

- create a set of resources
- control them as an unit

Kubernetes Operators

- It is a **pattern** that is becoming commonplace
  - There is *NOT* a specific API that you implement
  - You have to use the *Kubernetes API* anyway
- You define your own Resource
  - Defining new resources as **CRD** `Custom Resource Definitions`
  - Creating instances conforming to the CRD
    - that describes the *desidered state*
  - **Writing code that brings the system to this state**



Nuvolaris Architecture

Custom Resources Definitions

- Define your own Kubernetes Resources
  - Create new Kinds of resources
  - You can then create instances of this new Kind

Resource Handlers

- You need to write your own resource handler!
  - It responds to Kubernetes events
  - It interacts with Kubernetes APIs to perform operations

Components of a CRD

- Group, Kind and short names:
  - Example: `nuvolaris.org`, `Sample`, `sam`
- Spec and Status
  - Versioned
  - defined as an OpenApi Schema:

```
type: object
properties:
  spec:
    type: object
```

Defining a CRD (1/2)

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: samples.nuvolaris.org
spec:
  scope: Namespaced
  group: nuvolaris.org
  names:
    kind: Sample
    plural: samples
    singular: sample
    shortNames:
      - sam
```

Defining a CRD (2/2)

```
versions:
  - name: v1
    served: true
    storage: true
    subresources: { status: { } }
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            x-kubernetes-preserve-unknown-fields: true
          status:
            type: object
            x-kubernetes-preserve-unknown-fields: true
```

Instance

```yaml
apiVersion: nuvolaris.org/v1
kind: Sample
metadata:
  name: obj
spec:
  count: 2
```

Demo CRD

```
# check
cd lab
kubectl get nodes
kubectl apply -f demo-ns.yaml
kubectl config set-context --current --namespace demo
# create crd and instance
kubectl apply -f demo-crd.yaml
kubectl get crd
kubectl apply -f demo-obj.yaml
kubectl -n demo get sam
# cleanup
kubectl -n demo delete sam/obj
kubectl -n demo get sam
```

Coding an Operator

About `kopf`

- See kopf.readthedocs.io
- Python based
  - provided an handy `kopf` cli runner
- Handlers for the various Kubernetes events:
  - `@kopf.on.login`
  - `@kopf.on.create`
  - `@kopf.on.delete`
- It does not manage Kubernetes API

Login

- Kopf supports various autentication
  - Code to support either your `~/.kube/config` or the service token

```
@kopf.on.login()
def sample_login(**kwargs):
    token = '/var/run/secrets/kubernetes.io/serviceaccount/token'
    if os.path.isfile(token):
        logging.debug("found serviceaccount token: login via service account in kubernetes")
        return kopf.login_with_service_account(**kwargs)
    logging.debug("login via client")
    return  kopf.login_with_kubeconfig(**kwargs)
```

Handling object creation and deletion

```
@kopf.on.create('nuvolaris.org', 'v1', 'samples')
def sample_create(spec, **kwargs):
    print(spec)
    return { "message": "created" }
```

```
@kopf.on.delete('nuvolaris.org', 'v1', 'samples')
def sample_delete(spec, **kwargs):
    print(spec)
    return { "message": "delete" }
```

Handling creation of objects

```
# install and run kopf
poetry install
poetry run kopf
# run demo1.py
cat demo1.py
# run the operator specifying the namespace
poetry run kopf run -n demo demo1.py
# new terminal
kubectl apply -f demo-obj.yaml
kubectl -n demo get sam
kubectl delete -f demo-obj.yaml
```

Kustomize

Interacting with Kubernetes

- `kopf` does *not* provide how to interact with Kubernetes
  - You can use any other api like `pykube` or others
- We use... `kubectl` and `kustomize`
  - It may look "odd" to use an external command line tool
  - However, this allows compatibility with command line tools
    - avoding "strange" templating
    - easier development and debug

About `kustomize`

- Originally a separate tool, now part of `kubectl`
  - It works "customizing" sets of descriptors with rules
  - support many ways of *patching* the JSON/YAML
  - **NO TEMPLATING** (huge win over `helm`!)
- You simply do `kubectl apply -k <folder>`
  - It will search for `kustomization.yaml`
  - It will produce the output sent to Kubernetes
- Debug the output without applying with:
  `kubectl kustomize <folder>`

Simple `kustomizationl.yaml` with patch

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- demo-deployment.yaml
patches:
  - path: patch.yaml
```

- put it in a folder `deploy` and `apply -k deploy`

Sample patch of a `Deployment`

- We want to change the replica count

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deploy
  namespace: demo
spec:
  replicas: 1
```

- Intuitively, provide enough *context* to locate the descriptor
- Then, provide the **replaced fields**: `replicas: 1`

Demo Kustomize

```
cat demo-deployment.yaml
kubectl apply -f demo-deployment.yaml
kubectl get deploy ; kubectl get po
kubectl delete deploy demo-deploy
kubectl get deploy ; kubectl get po
# kustomize
cat kustomization.yaml
cat patch.yaml
# prepare the customization
rm -Rvf deploy ; mkdir deploy
cp demo-deployment.yaml kustomization.yaml patch.yaml deploy
# apply the kustomization
kubectl apply -k deploy
kubectl get deploy ; kubectl get po
kubectl delete -k deploy
kubectl get deploy ; kubectl get po
```

Implementing Operator

Implementing Operator

Using `kubectl` from the operator

```
# generate patch
def patch(n):
  return f"""apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deploy
spec:
  replicas: {n}
"""
# run kubectl
def kubectl(cmd, patch):
  with open(f"deploy/patch.yaml", "w") as f:
    f.write(patch)
  res = subprocess.run(["kubectl", cmd, "-k", "deploy"], capture_output=True)
  return res.stdout.decode()
```

Implementing the operator

```
@kopf.on.create('nuvolaris.org', 'v1', 'samples')
def sample_create(spec, **kwargs):
    count = spec["count"]
    message = kubectl("apply", patch(count))
    return { "message": message }

@kopf.on.delete('nuvolaris.org', 'v1', 'samples')
def sample_delete(spec, **kwargs):
    count = spec["count"]
    message = kubectl("delete", patch(count))
    return { "message": "delete" }
```

Demo Operator

```
# cleanup
kubectl delete -f demo-obj.yaml
poetry run kopf run -n demo demo2.py
# switch terminal
cat demo-obj.yaml
kubectl apply -f demo-obj.yaml
# checking if it worked
cat deploy/patch.yaml
kubectl get deploy ; kubectl get po
```

Packaging

- Create a Dockerfile embedding the operator
  - You need `poetry`, `kopf` an `kubectl` in the image
- Deploy the POD with the right permissions
  - You need to setup *Kuberbetes RBAC*
  - `ServiceAccount` and `ClusterRoleBinding`
- See `nuvolaris/nuvolaris-operator` for an example