europython
Jul 23 - 26 2020 Online

EuroPython 2020

ADVANCED INFRASTRUCTURE MANAGEMENT IN KUBERNETES USING PYTHON

Presented by Gautam Prajapati
www.gautamprajapati.com

ABOUT MYSELF

GAUTAM PRAJAPATI

www.gautamprajapati.com

- Software Engineer from Grofers India
- Bachelor's in Software Engineering from Delhi Technological University - Batch of 2018
- GSoC'17 fellow with LibreOffice - The Document Foundation
- Open source evangelist - Contributions to Mozilla(Firefox for Android), OpenMRS, FOSSASIA

TALK OVERVIEW

PHASE I - Introduction and Opportunities

- Problem scenarios from running applications on Kubernetes
- Configmap, Database cluster example
- Steps involved to run a celery cluster

PHASE II - Generalize Learnings and Goals

- Pain points of managing stateful in K8s
- Goal for Celery operator
- Extension Capabilities in K8s(CRDs and custom controllers)

PHASE III - Custom Controller & Demo

- Build controller incrementally to automate setup of Celery cluster
- Create custom resource and see the operator reacting to events
- Autoscale workers on queue length

PHASE IV - Conclusion and Q&A

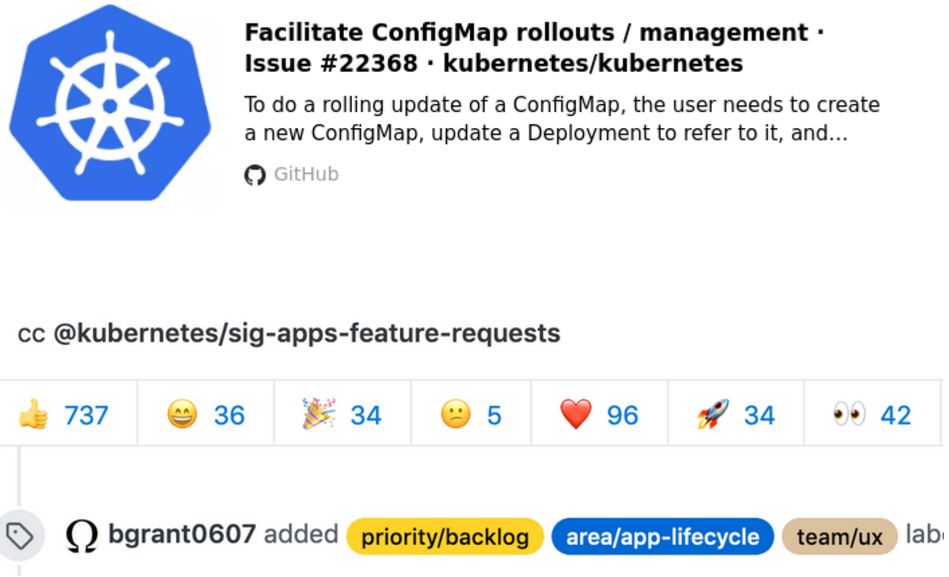
- Existing Operators, Frameworks and SDKs
- Other use cases
- Q&A

PROBLEMS

Real world scenarios coming from running applications on K8s

I. Common problem with configuration management in Kubernetes using ConfigMap and Secrets

- Need of restarting the deployment when a value is modified
- Imagine a watcher pod was managing those config objects and applications and it triggered the relevant deployments whenever config values changed

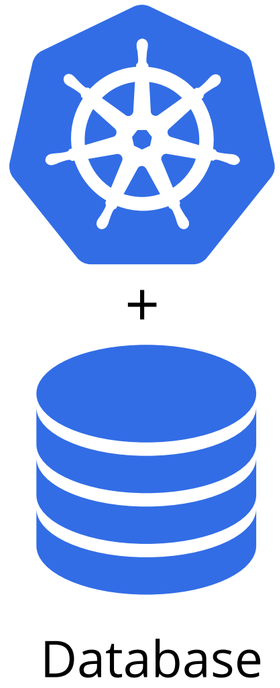


PROBLEMS

Real world scenarios coming from running applications on K8s

II. Setting Up a Database Cluster

- Running a database is easy(Deployment + PersistentVolume)
- Managing the cluster is difficult
  - Connection Pooling
  - Resize or Upgrades
  - Reconfiguration - Understanding of internals, tedius generation, templating and so-on
  - Backups - Coordination among different instances
  - Recovery - Restore from backup, rejoin cluster



*Example ref from - Automating stateful applications with operators by Josh Wood, Ryan Jarvinen - RedHat

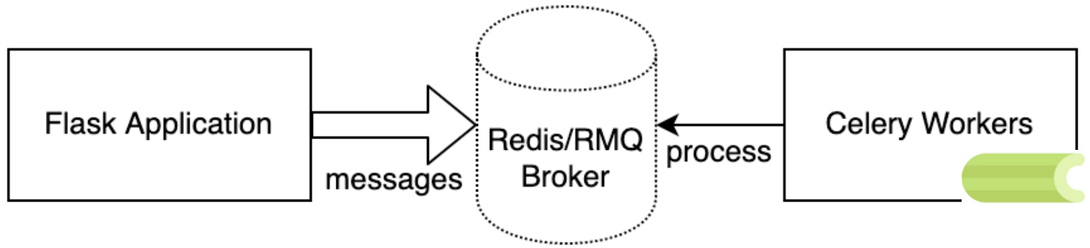
PROBLEMS

Manual steps for setting up celery cluster in K8s

III. Celery will be the focus of this talk

What is Celery?

- Popular distributed task queue system
- Typical Usecases - Asynchronous workloads like sending emails, sms, doing anything post request lifecycle, retries, etc.




```python
flask_app = Flask(__name__)
flask_app.config.update(
    CELERY_BROKER_URL='redis://redis-master/1',
    CELERY_RESULT_BACKEND='redis://redis-master/1'
)
celery_app = make_celery(flask_app)


@celery_app.task()
def add(a, b):
    return a + b
```

```
$ celery --app=app.celery_app worker
```

SETUP CELERY CLUSTER

What all needs to be done to make simple flask-celery example live on production?

- Write a celery **worker deployment** yaml, run it using **kubectl apply -f worker-dep.yaml**
- **Setup monitoring** - De-facto standard is **flower**
  - Write flower deployment spec
  - Expose a flower service
- **Setup autoscaling** configuration
  - Might want to scale number of workers on resource constraints
  - or queue length which isn't supported directly

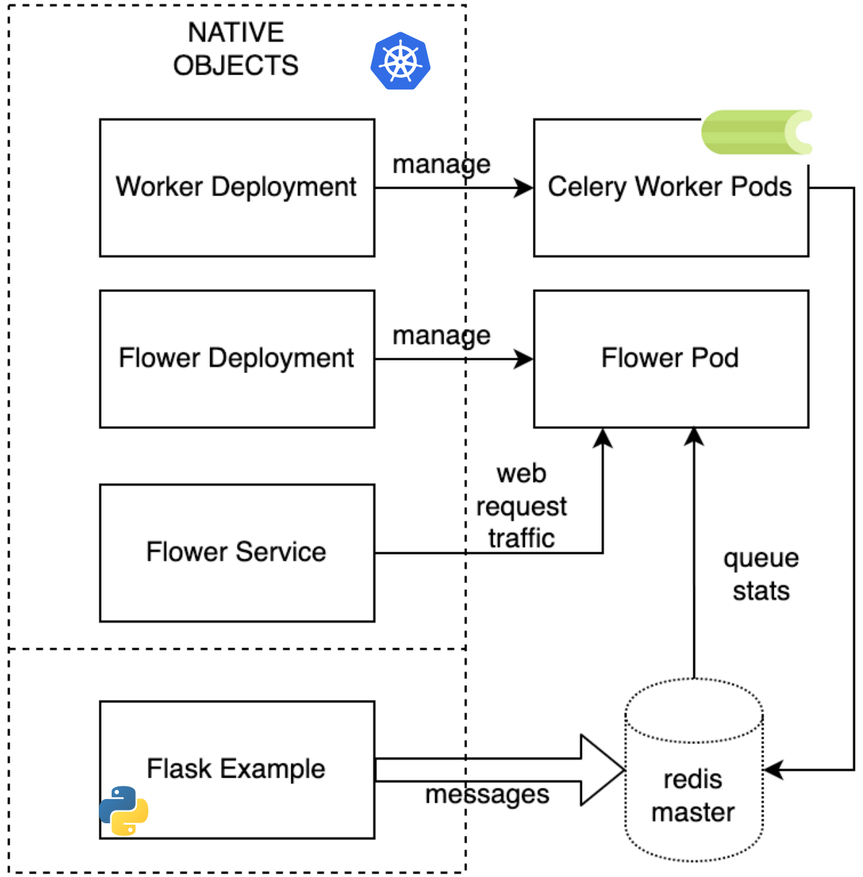
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: celery-worker
  name: celery-worker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: celery-worker
  template:
    metadata:
      labels:
        app: celery-worker
    spec:
      containers:
      - name: celery
        image: flask-celery-app
        imagePullPolicy: Never
        command: ["celery"]
        args:
        - "--app=app:celery_app"
        - "worker"
        - "--queues=default"
        - "--loglevel=info"
        - "--concurrency=2"
        resources:
        ...
```

SUMMARIZING

Running a celery cluster on production

Summarizing the problems -

- Not easy to get a new setup right
- Manual steps involved - Creating a deployment for workers, flower for monitoring, HPA for scaling etc.
- No way to setup multiple clusters in a consistent way
  - Everyone configures their own way
  - Possibilities of misconfiguration
  - Problems with infra audit, harder to manage

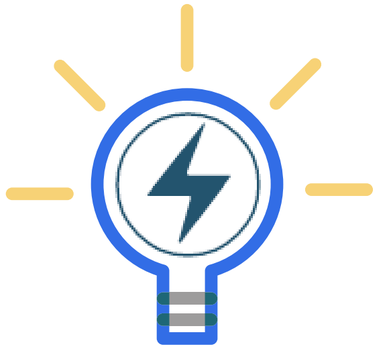


Typical Celery Cluster in Production

LEARNINGS

Managing stateless on Kubernetes is easy, Stateful is difficult

- Stateless application management(creation, scaling and recovery) is supported out of the box in K8s
- Stateful applications like databases, caching systems, message queuing systems need domain knowledge of handling how they are to be setup, scaled, upgraded and recovered properly for a business use-case
- Kubernetes is designed for automation. It is possible to extend it's behaviour to manage complex infrastructure, while staying in Python ecosystem
- Need to bridge the gap between application engineers and infrastructure operators who manually manage the services



THE GOAL

Deploying and managing stateful software can and should be made easy for everyone

I'm an Applications Developer, for my new celery cluster I want to -

- Provide parameters I care about in a standard Kubernetes yaml specification, edit them on production whenever I want
- Do nothing more than a simple **kubectl apply -f my-spec.yaml**
- Setup worker deployments and their monitoring automagically in the best way possible

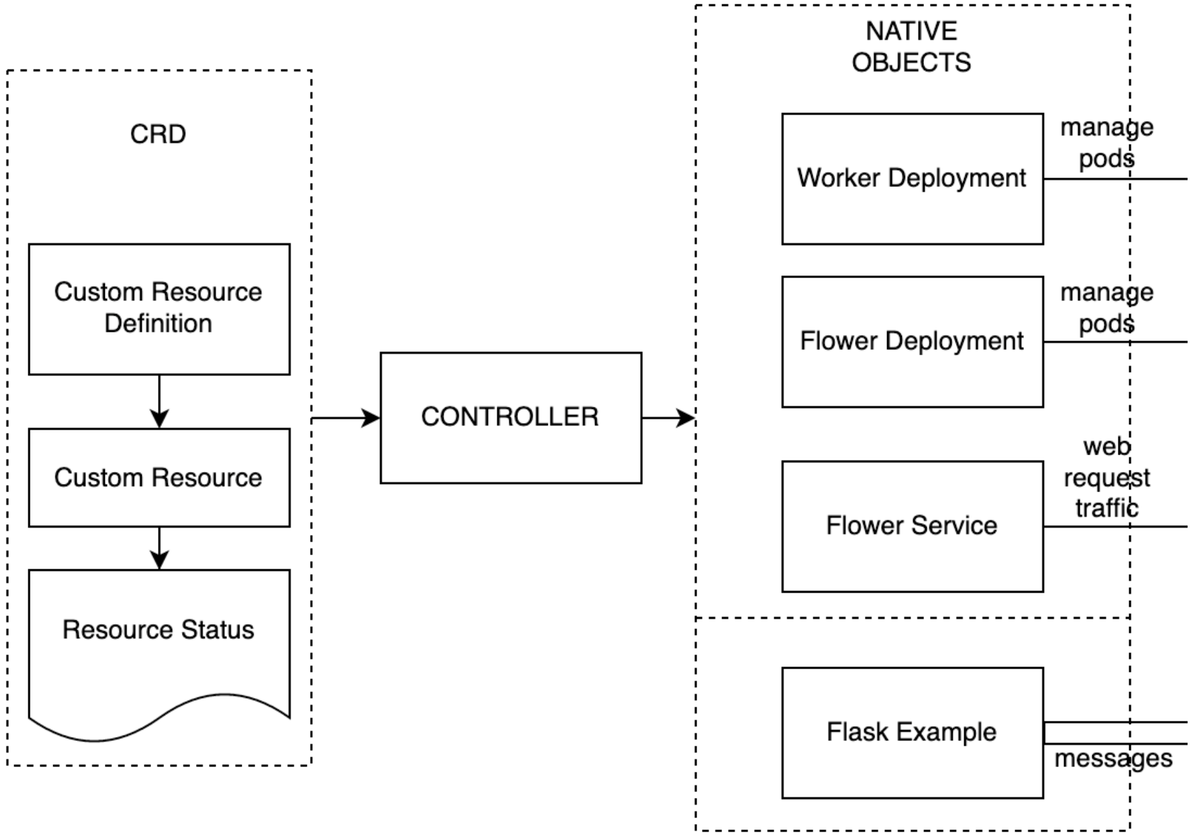
```yaml
apiVersion: celeryproject.org/v1alpha1
kind: Celery
metadata:
  name: example-celery-obj
spec:
  common:
    appName: celery-crd-example
    celeryApp: 'app:celery_app'
    image: example-image
  workerSpec:
    numOfWorkers: 2
    queues: celery # default queue name
    logLevel: debug
    concurrency: 2
    maxTasksPerChild: 100
    resources:
      ...
  flowerSpec:
    replicas: 1
    resources:
      ...
  scaleTargetRef:
  - kind: worker
    minReplicas: 2
    maxReplicas: 5
    metrics:
    - name: message_queue
      target:
        type: length
        averageValue: 100
```

custom-resource.yaml

CUSTOM RESOURCE DEFINITIONS(CRD)

Extending Kubernetes API using CRDs

- To make Kubernetes understand our custom resource named Celery
- Let's you define a structured schema of custom object
- Helps in standardising specification for managing multiple application instances in your Kubernetes cluster



CELERY CRD

Let's see at how a simple Celery CRD should look

```
$ kubectl apply -f deploy/crd.yaml
```

```
$ kubectl get crds
NAME                          CREATED AT
celery.celeryproject.org      2020-07-16T10:55:13
```

```
$ kubectl get celery
NAME                  CHILDREN   STATUS    AGE
example-celery-obj    3          CREATED   3d21h
```

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: celery.celeryproject.org
spec:
  scope: Namespaced
  group: celeryproject.org
  names:
    kind: Celery
    listKind: CeleryList
    plural: celery
    singular: celery
    shortNames:
      - cel
      - capp
  versions:
    - name: v1alpha1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          required: ["spec"]
          properties:
            ...
```

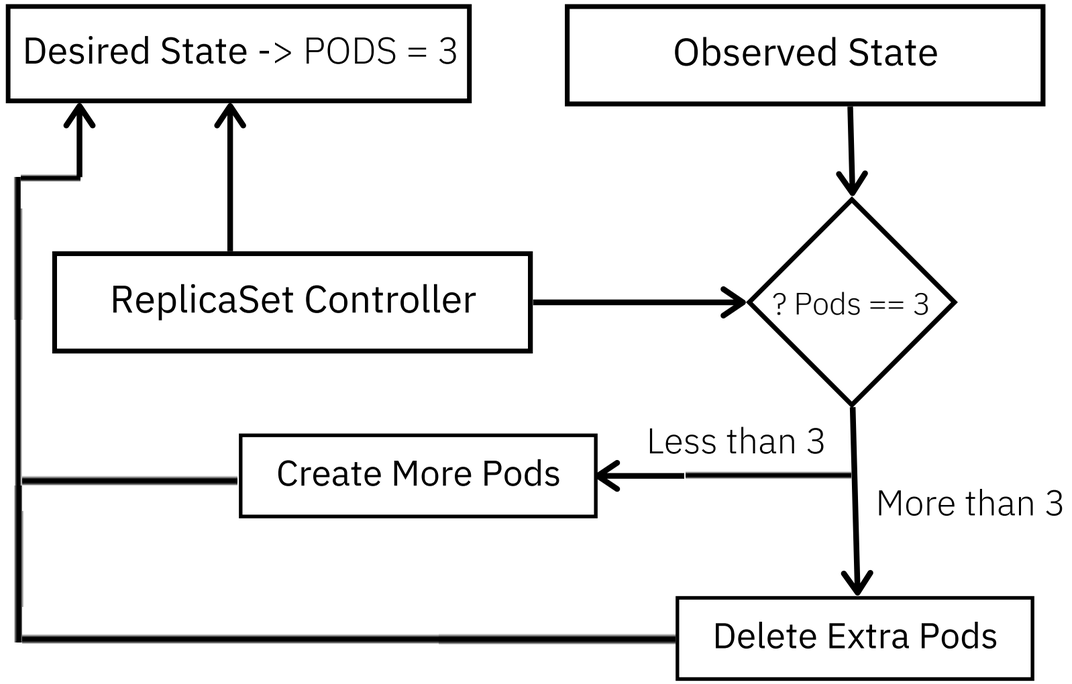
CONTROLLERS

Are at the core of self-healing capabilities of Kubernetes

- Execute control loops to manage the API objects they are watching
- Native examples - Deployment Controller, Replica Set Controller etc.
- Custom controllers can be written to watch and manage custom resources(CRDs)
- Celery CRD needs a controller maintain the desired spec provided by infra user

RECONCILIATION LOOPS

Actively try to match the desired state of a given object specified by cluster user to the currently observed state



Control loop for replicaset controller. Not an accurate representation, just for understanding purpose

OPERATOR PATTERN

Automating the work of a human operator in Kubernetes

WHAT ARE OPERATORS?

- Generally contain a CRD with custom controller implementation which takes care of creating, scaling, upgrades, recovery and more
- Software that extends native K8s abilities to reliably manage complex applications
- They can be called Kubernetes native apps
- All operators are controllers but not every controller is operator

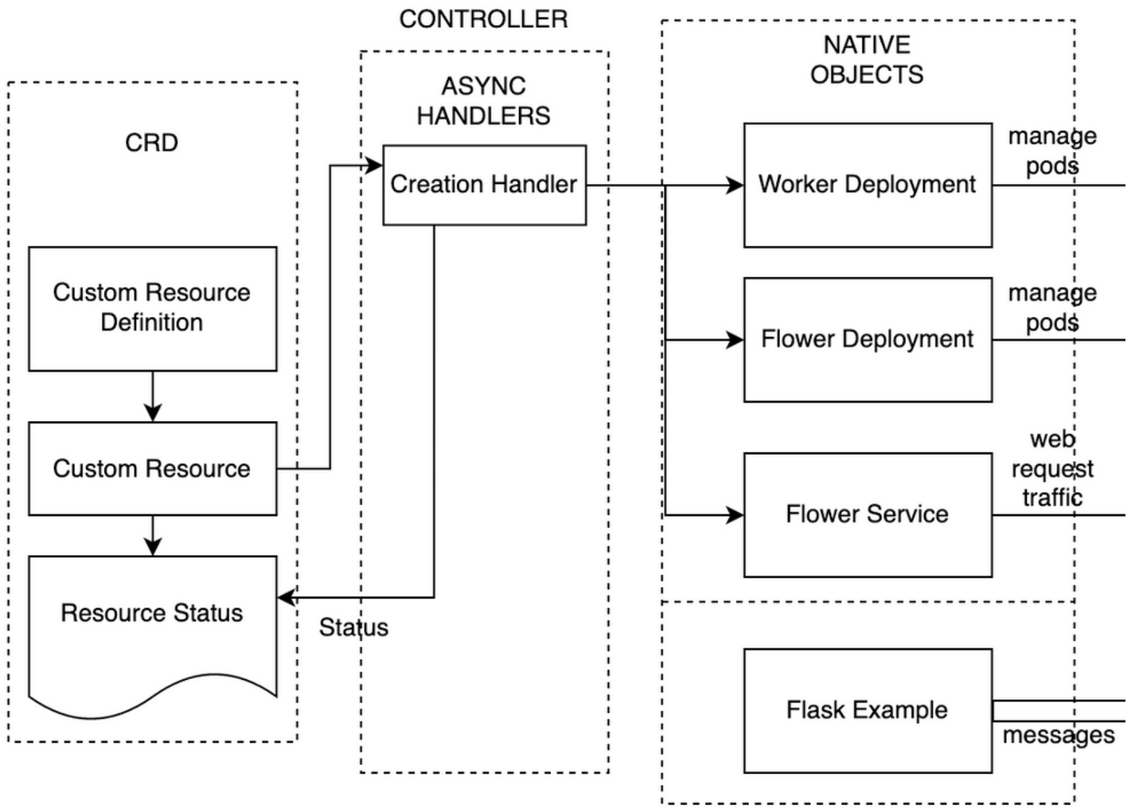
IMPLEMENTATION

- Operators can be written in any language/runtime which can act as a client for the Kubernetes API
- This talk also aims to encourage writing operators and supporting frameworks, in the Python ecosystem
- Currently Golang is a popular choice

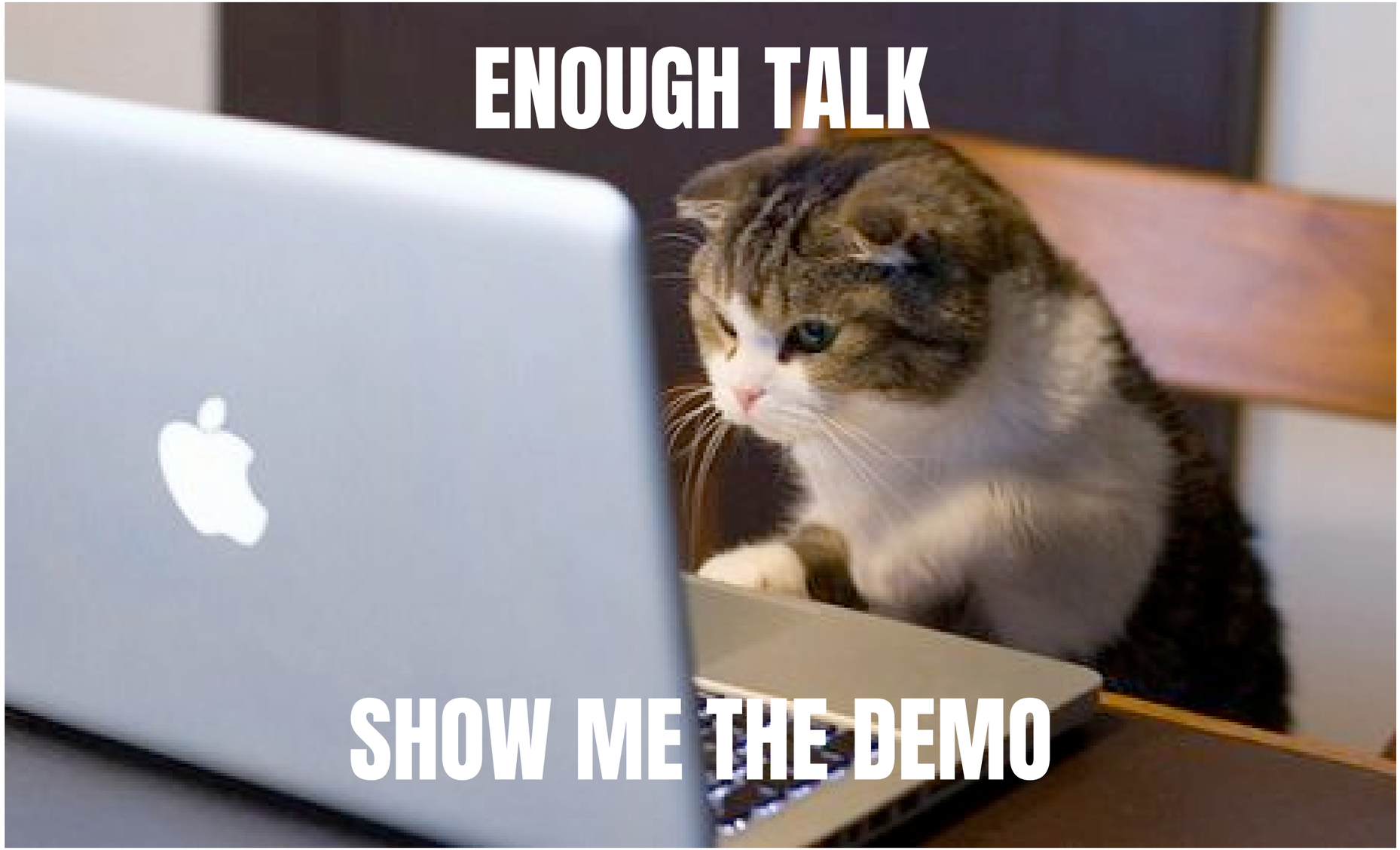
EXAMPLES

CREATION HANDLER

Handler taking care of creating a new celery cluster based on custom spec provided

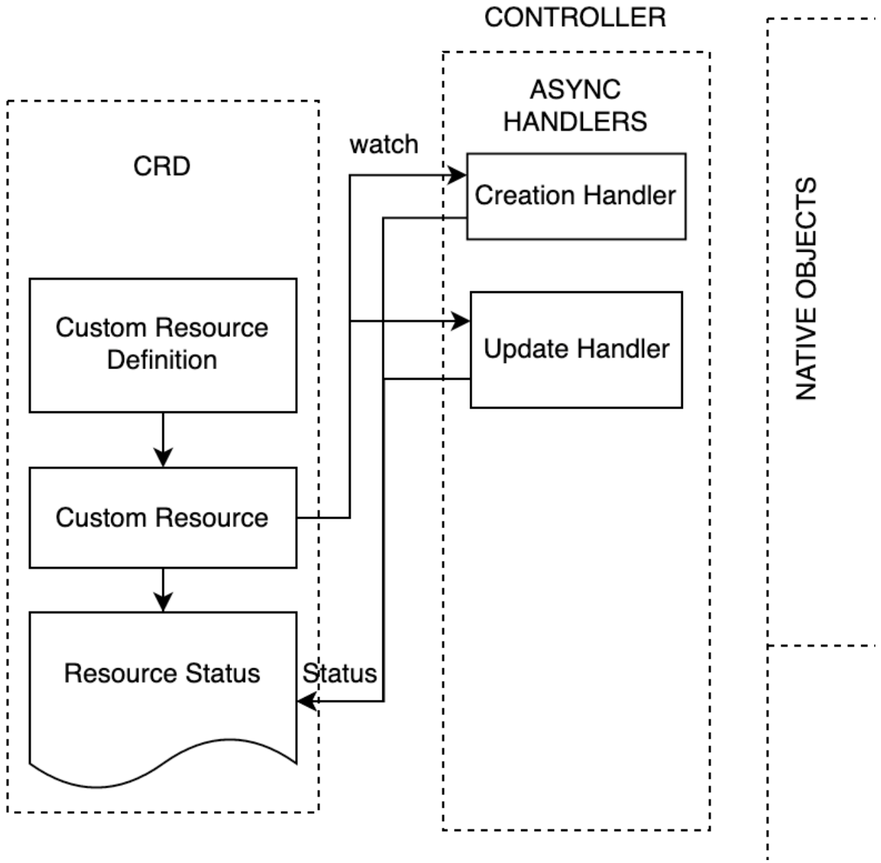



```python
@kopf.on.create('celeryproject.org', 'v1alpha1', 'celery')
def create_fn(spec, name, namespace, logger, **kwargs):
    # 1. Validation of spec
    val, err_msg = validate_spec(spec)
    if err_msg:
        status = 'Failed validation'
        raise kopf.PermanentError(f"{err_msg}. Got {val}")

    api = kubernetes.client.CoreV1Api()
    apps_api_instance = kubernetes.client.AppsV1Api()

    # 2. Deployment for celery workers
    worker_deployment = deploy_celery_workers(
        apps_api_instance, namespace, spec, logger
    )
    # 3. Deployment for flower
    flower_deployment = deploy_flower(
        apps_api_instance, namespace, spec, logger
    )
    # 4. Expose flower service
    flower_svc = expose_flower_service(
        api, namespace, spec, logger
    )
    children = [
        {
            'name': worker_deployment.metadata.name,
            'replicas': worker_deployment.spec.replicas,
            'kind': constants.DEPLOYMENT_KIND,
            'type': constants.WORKER_TYPE
        },
        ....
    ]

    return {
        'children': children,
        'children_count': len(children),
        'status': constants.STATUS_CREATED
    }
```


ENOUGH TALK
SHOW ME THE DEMO

UPDATION HANDLER

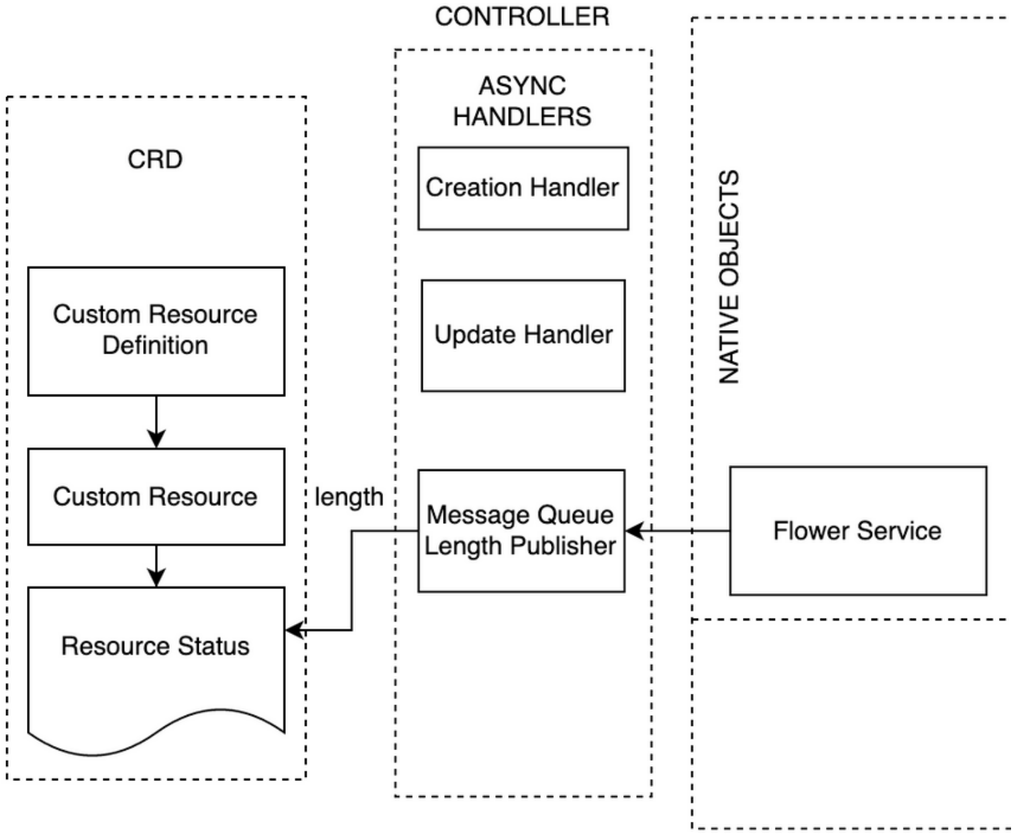
Handler taking care of updating the running celery cluster children




```python
@kopf.on.update('celeryproject.org', 'v1alpha1', 'celery')
def update_fn(spec, status, namespace, logger, **kwargs):
    diff = kwargs.get('diff')
    modified_spec = get_modified_spec_object(diff)

    api = kubernetes.client.CoreV1Api()
    apps_api_instance = kubernetes.client.AppsV1Api()

    if modified_spec.common_spec:
        # if common spec was updated, need to update every child
        return update_all_deployments(
            api, apps_api_instance, spec, status, namespace
        )
    else:
        result = status.get('update_fn') or status.get('create_fn')
        if modified_spec.worker_spec:
            # if worker spec was updated, update worker deployment
            worker_deployment = update_worker_deployment(
                apps_api_instance, spec, status, namespace
            )
            result = update_deployment_status(worker_deployment)

        if modified_spec.flower_spec:
            # if flower spec was updated, update flower deployment
            flower_deployment = update_flower_deployment(
                apps_api_instance, spec, status, namespace
            )
            result = update_deployment_status(flower_deployment)
        return result
```

QUEUE LENGTH PUBLISHER

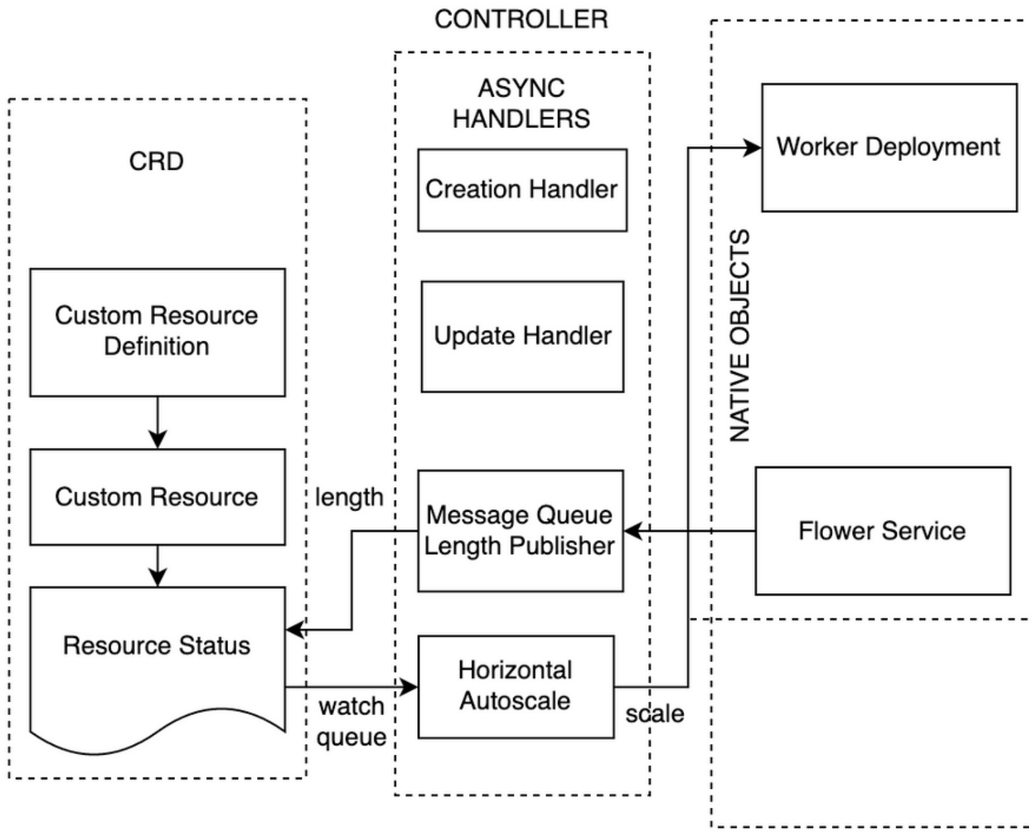
Handler publishing queue length every x seconds




```python
@kopf.timer('celeryproject.org', 'v1alpha1', 'celery',
            initial_delay=5, interval=10, idle=10)
def message_queue_length(spec, status, **kwargs):
    flower_svc_host = get_flower_svc_host(status)
    if not flower_svc_host:
        return

    url = f"http://{flower_svc_host}/api/queues/length"
    response = requests.get(url=url)
    if response.status_code == 200:
        return response.json().get('active_queues')

    return {
        "queue_length": 0
    }
```

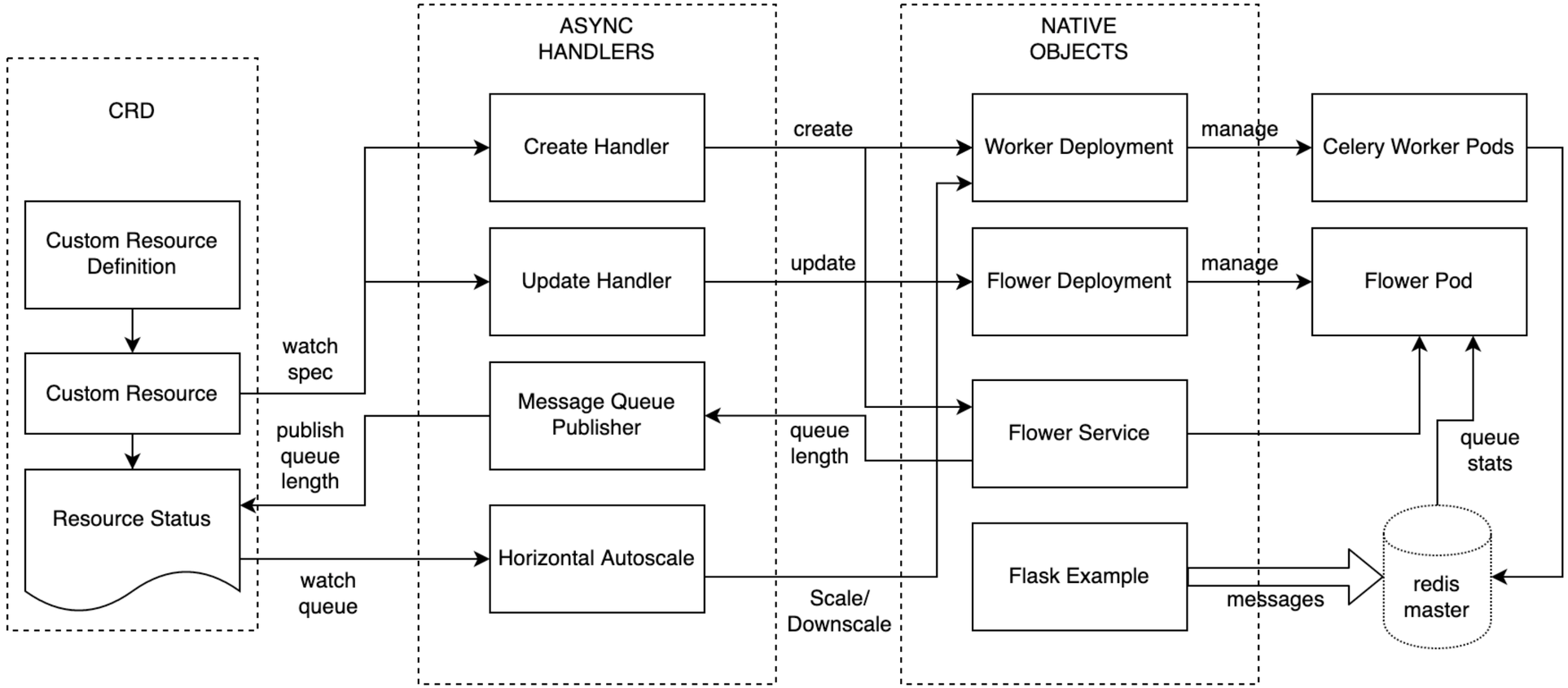
AUTOSCALE HANDLER

Handler taking care increasing/decreasing num of workers based on queue length




```python
@kopf.on.field('celeryproject.org', 'v1alpha1', 'celery',
               field='status.message_queue_length')
def horizontal_autoscale(spec, status, namespace, **kwargs):
    worker_dep_name = get_worker_dep_name(status)
    current_replicas = get_curr_replicas(worker_dep_name, status)
    updated_num_of_replicas = current_replicas
    scaling_targets = spec['scaleTargetRef']
    for scaling_target in scaling_targets:
        # For now we only support 1 i.e message queue length
        if scaling_target.get('kind') == 'worker':
            min_replicas = scaling_target.get(
              'minReplicas', spec['workerSpec']['numOfWorkers']
            )
            max_replicas = scaling_target.get('maxReplicas')
            queue_name = spec['workerSpec']['queues']
            current_queue_length = get_current_queue_len(
              queue_name, status
            )
            updated_num_of_replicas = calculate_updated_num_replicas(
                min_replicas, max_replicas, current_queue_length
            )

    patch_body = {
        "spec": {
            "replicas": updated_num_of_replicas,
        }
    }

    apps_api_instance = kubernetes.client.AppsV1Api()
    updated_deployment = apps_api_instance.patch_namespaced_deployment(
        worker_deployment_name, namespace, patch_body
    )

    return {
        'deploymentName': updated_deployment.metadata.name,
        'updated_num_of_replicas': updated_num_of_replicas
    }
```

CELERY OPERATOR ARCHITECTURE(POC)



SUMMARY

What all we talked about?

- Problems/Opportunities from running stateful apps on K8s
- Manual steps involving production celery cluster setup
- Goals for the Celery operator, Celery CRD and CR
- Controllers and Operator Pattern
- Creation Handler
- Updation Handler
- Autoscaling Implementation

NEXT STEPS

For the celery operator project - Github://brainbreaker/Celery-Kubernetes-Operator

- Some way to go for making it production ready, contributions/suggestions to improve are welcome
- Committing certain number of hours weekly to maintain the project based on feedback
- North Star aim would be to try and include it with Celery 5 release milestone of Dec 2020

CONCLUSION

WHAT ARE PEOPLE DOING WITH OPERATORS?

- Awesome Operators in the Wild - github://operator-framework/awesome-operators
- Registry for Operators - operatorhub.io
- Prometheus, Airflow, Couchbase, MongoDB, Consul, Vault, Zookeeper, Grafana, Cassandra, Postgres, AWS etc.
- !Idea - Operator to set up any new microservice, inject standard pieces like containers, volumes, logging, monitoring, Grafana dashboard, Newrelic etc. automatically

FRAMEWORKS AND RESOURCES TO BUILD OPERATORS

- Kubernetes Operator Pythonic Framework (Kopf)
- Operator-SDK (Golang) - SDK for building Kubernetes applications
- Kubebuilder(Golang) - https://kubebuilder.io
- Metacontroller - Makes it easier to write custom controllers in any language

Q&A

Shoot your questions

 28gautam97  brainbreaker  brainbreaker  28gautam97@gmail.com