Kopf

Sergey Vasilyev

Apr 08, 2024

FIRST STEPS:

CHAPTER

ONE

INSTALLATION

Prerequisites:

- Python >= 3.8 (CPython and PyPy are officially tested and supported).

To install Kopf:

```
pip install kopf
```

If you use some of the managed Kubernetes services which require a sophisticated authentication beyond username+password, fixed tokens, or client SSL certs (also see *authentication piggy-backing*):

```
pip install kopf[full-auth]
```

If you want extra i/o performance under the hood, install it as (also see *Custom event loops*):

```
pip install kopf[uvloop]
```

Unless you use the standalone mode, create a few Kopf-specific custom resources in the cluster:

```
kubectl apply -f https://github.com/nolar/kopf/raw/main/peering.yaml
```

Optionally, if you are going to use the examples or the code snippets:

```
kubectl apply -f https://github.com/nolar/kopf/raw/main/examples/crd.yaml
```

You are ready to go:

```
kopf --help
kopf run --help
kopf run examples/01-minimal/example.py
```

CHAPTER

TWO

CONCEPTS

Kubernetes is a container orchestrator.

It provides some basic primitives to orchestrate application deployments on a low level —such as the pods, jobs, deployments, services, ingresses, persistent volumes and volume claims, secrets— and allows a Kubernetes cluster to be extended with the arbitrary custom resources and custom controllers.

On the top level, it consists of the Kubernetes API, through which the users talk to Kubernetes, internal storage of the state of the objects (etcd), and a collection of controllers. The command-line tooling (`kubectl`) can also be considered as a part of the solution.

The **Kubernetes controller** is the logic (i.e. the behaviour) behind most objects, both built-in and added as extensions of Kubernetes. Examples of objects are ReplicaSet and Pods, created when a Deployment object is created, with the rolling version upgrades, and so on.

The main purpose of any controller is to bring the actual state of the cluster to the desired state, as expressed with the resources/object specifications.

The **Kubernetes operator** is one kind of the controllers, which orchestrates objects of a specific kind, with some domain logic implemented inside.

The essential difference between operators and the controllers is that operators are domain-specific controllers, but not all controllers are necessary operators: for example, the built-in controllers for pods, deployments, services, etc, so as the extensions of the object's life-cycles based on the labels/annotations, are not operators, but just controllers.

The essential similarity is that they both implement the same pattern: watching the objects and reacting to the objects' events (usually the changes).

Kopf is a framework to build Kubernetes operators in Python.

Like any framework, Kopf provides both the "outer" toolkit to run the operator, to talk to the Kubernetes cluster, and to marshal the Kubernetes events into the pure-Python functions of the Kopf-based operator, and the "inner" libraries to assist with a limited set of common tasks of manipulating the Kubernetes objects (however, it is not yet another Kubernetes client library).

See also:

See *Architecture* to understand how Kopf works in detail, and what it does exactly.

See *Vision* and *Alternatives* to understand Kopf's self-positioning in the world of Kubernetes.

See also:

- https://en.wikipedia.org/wiki/Kubernetes

- https://coreos.com/operators/
- https://stackoverflow.com/a/47857073
- https://github.com/kubeflow/tf-operator/issues/300

CHAPTER

THREE

SAMPLE PROBLEM

Throughout this user documentation, we try to solve a little real-world problem with Kopf, step by step, presenting and explaining all the Kopf features one by one.

3.1 Problem Statement

In Kubernetes, there are no ephemeral volumes of big sizes, e.g. 500 GB. By ephemeral, it means that the volume does not persist after it is used. Such volumes can be used as a workspace for large data-crunching jobs.

There is Local Ephemeral Storage, which allocates some space on a node's root partition shared with the docker images and other containers, but it is often limited in size depending on the node/cluster config:

```yaml
kind: Pod
spec:
  containers:
    - name: main
      resources:
        requests:
          ephemeral-storage: 1G
        limits:
          ephemeral-storage: 1G
```

There is a PersistentVolumeClaim resource kind, but it is persistent, i.e. not deleted after they are created (only manually deletable).

There is StatefulSet, which has the volume claim template, but the volume claim is again persistent, and the set does not follow the same flow as the Jobs do, more like the Deployments.

3.2 Problem Solution

We will implement the `EphemeralVolumeClaim` object kind, which will be directly equivalent to `PersistentVolumeClaim` (and will use it internally), but with a little extension:

It will be *designated* for a pod or pods with specific selection criteria.

Once used, and all those pods are gone and are not going to be restarted, the ephemeral volume claim will be deleted after a *grace period*.

For safety, there will be an *expiry period* for the cases when the claim was not used: e.g. if the pod could not start for some reasons so that the claim does not remain stale forever.

The lifecycle of an `EphemeralVolumeClaim` is this:

- Created by a user with a template of `PersistentVolumeClaim` and a designated pod selector (by labels).
- Waits until the claim is used at least once.
  - At least for N seconds of the safe time to allow the pods to start.
  - At most for M seconds for the case when the pod has failed to start, but the claim was created.
- Deletes the `PersistentVolumeClaim` after either the pod is finished, or the wait time has elapsed.

See also:

This documentation only highlights the main patterns & tricks of Kopf, but does not dive deep into the implementation of the operator's domain. The fully functional solution for `EphemeralVolumeClaim` resources, which is used for this documentation, is available at the following link:

- https://github.com/nolar/ephemeral-volume-claims

CHAPTER

FOUR

ENVIRONMENT SETUP

We need a running Kubernetes cluster and some tools for our experiments. If you have a cluster already preconfigured, you can skip this section. Otherwise, let's install minikube locally (e.g. for MacOS):

- Python >= 3.8 (running in a venv is recommended, though is not necessary).
- Install kubectl
- *Install minikube* (a local Kubernetes cluster)
- *Install Kopf*

Warning: Unfortunately, Minikube cannot handle the PVC/PV resizing, as it uses the HostPath provider internally. You can either skip the *Updating the objects* step of this tutorial (where the sizes of the volumes are changed), or you can use an external Kubernetes cluster with real dynamically sized volumes.

CHAPTER

FIVE

CUSTOM RESOURCES

5.1 Custom Resource Definition

Let us define a CRD (custom resource definition) for our object:

Listing 1: crd.yaml

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: ephemeralvolumeclaims.kopf.dev
spec:
  scope: Namespaced
  group: kopf.dev
  names:
    kind: EphemeralVolumeClaim
    plural: ephemeralvolumeclaims
    singular: ephemeralvolumeclaim
    shortNames:
      - evcs
      - evc
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              x-kubernetes-preserve-unknown-fields: true
            status:
              type: object
              x-kubernetes-preserve-unknown-fields: true
```

Note the short names: they can be used as the aliases on the command line, when getting a list or an object of that kind.

And apply the definition to the cluster:

```
kubectl apply -f crd.yaml
```

If you want to revert this operation (e.g., to try it again):

```
kubectl delete crd ephemeralvolumeclaims.kopf.dev
kubectl delete -f crd.yaml
```

5.2 Custom Resource Objects

Now, we can already create the objects of this kind, apply it to the cluster, modify and delete them. Nothing will happen, since there is no implemented logic behind the objects yet.

Let's make a sample object:

Listing 2: obj.yaml

```yaml
apiVersion: kopf.dev/v1
kind: EphemeralVolumeClaim
metadata:
  name: my-claim
```

This is the minimal yaml file needed, with no spec or fields inside. We will add them later.

Apply it to the cluster:

```
kubectl apply -f obj.yaml
```

Get a list of the existing objects of this kind with one of the commands:

```
kubectl get EphemeralVolumeClaim
kubectl get ephemeralvolumeclaims
kubectl get ephemeralvolumeclaim
kubectl get evcs
kubectl get evc
```

Please note that we can use the short names as specified on the custom resource definition.

See also:

- kubectl imperative style (create/edit/patch/delete)
- kubectl declarative style (apply)

CHAPTER

SIX

STARTING THE OPERATOR

Previously, we have defined a *problem* that we are solving, and created the *custom resource definitions* for the ephemeral volume claims.

Now, we are ready to write some logic for this kind of objects. Let's start with an operator skeleton that does nothing useful – just to see how it can be started.

Listing 1: ephemeral.py

```
import kopf
import logging

@kopf.on.create('ephemeralvolumeclaims')
def create_fn(body, **kwargs):
    logging.info(f"A handler is called with body: {body}")
```

Note: Despite an obvious desire, do not name the file as `operator.py`, since there is a built-in module in Python 3 with this name, and there could be potential conflicts on the imports.

Let's run the operator and see what will happen:

```
kopf run ephemeral.py --verbose
```

The output looks like this:

```
[2019-05-31 10:42:11,870] kopf.config          [DEBUG   ] configured via kubeconfig file
[2019-05-31 10:42:11,913] kopf.reactor.peering [WARNING ] Default peering object is not␣
↪found, falling back to the standalone mode.
[2019-05-31 10:42:12,037] kopf.reactor.handlin [DEBUG   ] [default/my-claim] First␣
↪appearance: {'apiVersion': 'kopf.dev/v1', 'kind': 'EphemeralVolumeClaim', 'metadata': {
↪'annotations': {'kubectl.kubernetes.io/last-applied-configuration': '{"apiVersion":
↪"kopf.dev/v1","kind":"EphemeralVolumeClaim","metadata":{"annotations":{},"name":"my-
↪claim","namespace":"default"}}\n'}, 'creationTimestamp': '2019-05-29T00:41:57Z',
↪'generation': 1, 'name': 'my-claim', 'namespace': 'default', 'resourceVersion': '47720
↪', 'selfLink': '/apis/kopf.dev/v1/namespaces/default/ephemeralvolumeclaims/my-claim',
↪'uid': '904c2b9b-81aa-11e9-a202-a6e6b278a294'}}
[2019-05-31 10:42:12,038] kopf.reactor.handlin [DEBUG   ] [default/my-claim] Adding the␣
↪finalizer, thus preventing the actual deletion.
[2019-05-31 10:42:12,038] kopf.reactor.handlin [DEBUG   ] [default/my-claim] Patching␣
↪with: {'metadata': {'finalizers': ['KopfFinalizerMarker']}}
[2019-05-31 10:42:12,165] kopf.reactor.handlin [DEBUG   ] [default/my-claim] Creation is␣
```

(continues on next page)

(continued from previous page)

```
↪in progress: {'apiVersion': 'kopf.dev/v1', 'kind': 'EphemeralVolumeClaim', 'metadata':
↪{'annotations': {'kubectl.kubernetes.io/last-applied-configuration': '{"apiVersion":
↪"kopf.dev/v1","kind":"EphemeralVolumeClaim","metadata":{"annotations":{},"name":"my-
↪claim","namespace":"default"}}\n'}, 'creationTimestamp': '2019-05-29T00:41:57Z',
↪'finalizers': ['KopfFinalizerMarker'], 'generation': 1, 'name': 'my-claim', 'namespace
↪': 'default', 'resourceVersion': '47732', 'selfLink': '/apis/kopf.dev/v1/namespaces/
↪default/ephemeralvolumeclaims/my-claim', 'uid': '904c2b9b-81aa-11e9-a202-a6e6b278a294'}
↪}
[2019-05-31 10:42:12,166] root                  [INFO    ] A handler is called with body:
↪{'apiVersion': 'kopf.dev/v1', 'kind': 'EphemeralVolumeClaim', 'metadata': {'annotations
↪': {'kubectl.kubernetes.io/last-applied-configuration': '{"apiVersion":"kopf.dev/v1",
↪"kind":"EphemeralVolumeClaim","metadata":{"annotations":{},"name":"my-claim","namespace
↪":"default"}}\n'}, 'creationTimestamp': '2019-05-29T00:41:57Z', 'finalizers': [
↪'KopfFinalizerMarker'], 'generation': 1, 'name': 'my-claim', 'namespace': 'default',
↪'resourceVersion': '47732', 'selfLink': '/apis/kopf.dev/v1/namespaces/default/
↪ephemeralvolumeclaims/my-claim', 'uid': '904c2b9b-81aa-11e9-a202-a6e6b278a294'}, 'spec
↪': {}, 'status': {}}
[2019-05-31 10:42:12,168] kopf.reactor.handlin [DEBUG   ] [default/my-claim] Invoking␣
↪handler 'create_fn'.
[2019-05-31 10:42:12,173] kopf.reactor.handlin [INFO    ] [default/my-claim] Handler
↪'create_fn' succeeded.
[2019-05-31 10:42:12,210] kopf.reactor.handlin [INFO    ] [default/my-claim] All␣
↪handlers succeeded for creation.
[2019-05-31 10:42:12,223] kopf.reactor.handlin [DEBUG   ] [default/my-claim] Patching␣
↪with: {'status': {'kopf': {'progress': None}}, 'metadata': {'annotations': {'kopf.
↪zalando.org/last-handled-configuration': '{"apiVersion": "kopf.dev/v1", "kind":
↪"EphemeralVolumeClaim", "metadata": {"name": "my-claim", "namespace": "default"}, "spec
↪": {}}'}}}
[2019-05-31 10:42:12,342] kopf.reactor.handlin [DEBUG   ] [default/my-claim] Updating is␣
↪in progress: {'apiVersion': 'kopf.dev/v1', 'kind': 'EphemeralVolumeClaim', 'metadata':
↪{'annotations': {'kopf.zalando.org/last-handled-configuration': '{"apiVersion": "kopf.
↪dev/v1", "kind": "EphemeralVolumeClaim", "metadata": {"name": "my-claim", "namespace":
↪"default"}, "spec": {}}', 'kubectl.kubernetes.io/last-applied-configuration': '{
↪"apiVersion":"kopf.dev/v1","kind":"EphemeralVolumeClaim","metadata":{"annotations":{},
↪"name":"my-claim","namespace":"default"}}\n'}, 'creationTimestamp': '2019-05-
↪29T00:41:57Z', 'finalizers': ['KopfFinalizerMarker'], 'generation': 2, 'name': 'my-
↪claim', 'namespace': 'default', 'resourceVersion': '47735', 'selfLink': '/apis/kopf.
↪dev/v1/namespaces/default/ephemeralvolumeclaims/my-claim', 'uid': '904c2b9b-81aa-11e9-
↪a202-a6e6b278a294'}, 'status': {'kopf': {}}}
[2019-05-31 10:42:12,343] kopf.reactor.handlin [INFO    ] [default/my-claim] All␣
↪handlers succeeded for update.
[2019-05-31 10:42:12,362] kopf.reactor.handlin [DEBUG   ] [default/my-claim] Patching␣
↪with: {'status': {'kopf': {'progress': None}}, 'metadata': {'annotations': {'kopf.
↪zalando.org/last-handled-configuration': '{"apiVersion": "kopf.dev/v1", "kind":
↪"EphemeralVolumeClaim", "metadata": {"name": "my-claim", "namespace": "default"}, "spec
↪": {}}'}}}
```

Note that the operator has noticed an object created before the operator was even started, and handled the object because it was not handled before.

Now, you can stop the operator with Ctrl-C (twice), and start it again:

```
kopf run ephemeral.py --verbose
```

The operator will not handle the object, as now it is already successfully handled. This is important in case the operator is restarted if it runs in a normally deployed pod, or when you restart the operator for debugging.

Let's delete and re-create the same object to see the operator reacting:

```
kubectl delete -f obj.yaml
kubectl apply -f obj.yaml
```

CHAPTER

SEVEN

CREATING THE OBJECTS

Previously (*Starting the operator*), we have created a skeleton operator and learned to start it and see the logs. Now, let's add a few meaningful reactions to solve our problem (*Sample Problem*).

We want to create a real `PersistentVolumeClaim` object immediately when an `EphemeralVolumeClaim` is created this way:

Listing 1: evc.yaml

```yaml
apiVersion: kopf.dev/v1
kind: EphemeralVolumeClaim
metadata:
  name: my-claim
spec:
  size: 1G
```

```
kubectl apply -f evc.yaml
```

First, let's define a template of the persistent volume claim (with the Python template string, so that no extra template engines are needed):

Listing 2: pvc.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: "{name}"
  annotations:
    volume.beta.kubernetes.io/storage-class: standard
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: "{size}"
```

Let's extend our only handler. We will use the official Kubernetes client library (`pip install kubernetes`):

Listing 3: ephemeral.py

```python
import os
import kopf
import kubernetes
```

(continues on next page)

(continued from previous page)

```
import yaml

@kopf.on.create('ephemeralvolumeclaims')
def create_fn(spec, name, namespace, logger, **kwargs):

    size = spec.get('size')
    if not size:
        raise kopf.PermanentError(f"Size must be set. Got {size!r}.")

    path = os.path.join(os.path.dirname(__file__), 'pvc.yaml')
    tmpl = open(path, 'rt').read()
    text = tmpl.format(name=name, size=size)
    data = yaml.safe_load(text)

    api = kubernetes.client.CoreV1Api()
    obj = api.create_namespaced_persistent_volume_claim(
        namespace=namespace,
        body=data,
    )

    logger.info(f"PVC child is created: {obj}")
```

And let us try it in action (assuming the operator is running in the background):

```
kubectl apply -f evc.yaml
```

Wait 1-2 seconds, and take a look:

```
kubectl get pvc
```

Now, the PVC can be attached to the pods by the same name, as EVC is named.

Note: If you have to re-run the operator and hit an HTTP 409 error saying "persistentvolumeclaims "my-claim" already exists", then remove it manually:

```
kubectl delete pvc my-claim
```

See also:

See also *Handlers*, *Error handling*, *Hierarchies*.

CHAPTER

EIGHT

UPDATING THE OBJECTS

Warning: Unfortunately, Minikube cannot handle the PVC/PV resizing, as it uses the HostPath provider internally. You can either skip this step of the tutorial, or you can use an external Kubernetes cluster with real dynamically sized volumes.

Previously (*Creating the objects*), we have implemented a handler for the creation of an `EphemeralVolumeClaim` (EVC), and created the corresponding `PersistantVolumeClaim` (PVC).

What will happen if we change the size of the EVC when it already exists? The PVC must be updated accordingly to match its parent EVC.

First, we have to remember the name of the created PVC: Let's extend the creation handler we already have from the previous step with one additional line:

Listing 1: ephemeral.py

```
@kopf.on.create('ephemeralvolumeclaims')
def create_fn(spec, name, namespace, logger, **kwargs):

    size = spec.get('size')
    if not size:
        raise kopf.PermanentError(f"Size must be set. Got {size!r}.")

    path = os.path.join(os.path.dirname(__file__), 'pvc.yaml')
    tmpl = open(path, 'rt').read()
    text = tmpl.format(size=size, name=name)
    data = yaml.safe_load(text)

    api = kubernetes.client.CoreV1Api()
    obj = api.create_namespaced_persistent_volume_claim(
        namespace=namespace,
        body=data,
    )

    logger.info(f"PVC child is created: {obj}")

    return {'pvc-name': obj.metadata.name}
```

Whatever is returned from any handler, is stored in the object's status under that handler id (which is the function name by default). We can see that with kubectl:

```
kubectl get -o yaml evc my-claim
```

```
spec:
  size: 1G
status:
  create_fn:
    pvc-name: my-claim
  kopf: {}
```

Note: If the above change causes `Patching failed with inconsistencies` debug warnings and/or your EVC YAML doesn't show a `.status` field, make sure you have set the `x-kubernetes-preserve-unknown-fields: true` field in your CRD on either the entire object or just the `.status` field as detailed in *Custom Resources*. Without setting this field, Kubernetes will prune the `.status` field when Kopf tries to update it. For more info on field pruning, see the Kubernetes docs.

Let's add a yet another handler, but for the "update" cause. This handler gets this stored PVC name from the creation handler, and patches the PVC with the new size from the EVC:

```
@kopf.on.update('ephemeralvolumeclaims')
def update_fn(spec, status, namespace, logger, **kwargs):

    size = spec.get('size', None)
    if not size:
        raise kopf.PermanentError(f"Size must be set. Got {size!r}.")

    pvc_name = status['create_fn']['pvc-name']
    pvc_patch = {'spec': {'resources': {'requests': {'storage': size}}}}

    api = kubernetes.client.CoreV1Api()
    obj = api.patch_namespaced_persistent_volume_claim(
        namespace=namespace,
        name=pvc_name,
        body=pvc_patch,
    )

    logger.info(f"PVC child is updated: {obj}")
```

Now, let's change the EVC's size:

```
kubectl edit evc my-claim
```

Or by patching it:

```
kubectl patch evc my-claim --type merge -p '{"spec": {"size": "2G"}}'
```

Keep in mind the PVC size can only be increased, never decreased.

Give the operator a few seconds to handle the change.

Check the size of the actual PV behind the PVC, which is now increased:

```
kubectl get pv
```

```
NAME                                       CAPACITY   ACCESS MODES   ...
pvc-a37b65bd-8384-11e9-b857-42010a800265   2Gi        RWO            ...
```

Warning: Kubernetes & `kubectl` improperly show the capacity of PVCs: it remains the same (1G) event after the change. The size of the actual PV (Persistent Volume) of each PVC is important! This issue is not related to Kopf, so we go around it.

CHAPTER

NINE

DIFFING THE FIELDS

Previously (*Updating the objects*), we have set the size of PVC to be updated every time the size of EVC is updated, i.e. the cascaded updates.

What will happen if the user re-labels the EVC?

```
kubectl label evc my-claim application=some-app owner=me
```

Nothing. The EVC update handler will be called, but it only uses the size field. Other fields are ignored.

Let's re-label the PVC with the labels of its EVC, and keep them in sync. The sync is one-way: re-labelling the child PVC does not affect the parent EVC.

9.1 Old & New

It can be done the same way as the size update handlers, but we will use another feature of Kopf to track one specific field only:

Listing 1: ephemeral.py

```
@kopf.on.field('ephemeralvolumeclaims', field='metadata.labels')
def relabel(old, new, status, namespace, **kwargs):

    pvc_name = status['create_fn']['pvc-name']
    pvc_patch = {'metadata': {'labels': new}}

    api = kubernetes.client.CoreV1Api()
    obj = api.patch_namespaced_persistent_volume_claim(
        namespace=namespace,
        name=pvc_name,
        body=pvc_patch,
    )
```

The *old* & *new* kwargs contain the old & new values of the field (or of the whole object for the object handlers).

It will work as expected when the user adds new labels and changes the existing labels, but not when the user deletes the labels from the EVC.

Why? Because of how patching works in Kubernetes API: it *merges* the dictionaries (with some exceptions). To delete a field from the object, it should be set to `None` in the patch object.

So, we should know which fields were deleted from EVC. Natively, Kubernetes does not provide this information for the object events, since Kubernetes notifies the operators only with the newest state of the object – as seen in *body*/*meta* kwargs.

9.2 Diffs

Kopf tracks the state of the objects and calculates the diffs. The diffs are provided as the *diff* kwarg; the old & new states of the object or field – as the *old* & *new* kwargs.

A diff-object has this structure:

```
((action, n-tuple of object or field path, old, new),)
```

with example:

```
(('add', ('metadata', 'labels', 'label1'), None, 'new-value'),
 ('change', ('metadata', 'labels', 'label2'), 'old-value', 'new-value'),
 ('remove', ('metadata', 'labels', 'label3'), 'old-value', None),
 ('change', ('spec', 'size'), '1G', '2G'))
```

For the field-handlers, it will be the same, but the field path will be relative to the handled field, and unrelated fields will be filtered out. For example, if the field is `metadata.labels`:

```
(('add', ('label1',), None, 'new-value'),
 ('change', ('label2',), 'old-value', 'new-value'),
 ('remove', ('label3',), 'old-value', None))
```

Now, let's use this feature to explicitly react to the re-labelling of the EVCs. Note that the `new` value for the removed dict key is `None`, exactly as needed for the patch object (i.e. the field is present there):

Listing 2: ephemeral.py

```
@kopf.on.field('ephemeralvolumeclaims', field='metadata.labels')
def relabel(diff, status, namespace, **kwargs):

    labels_patch = {field[0]: new for op, field, old, new in diff}
    pvc_name = status['create_fn']['pvc-name']
    pvc_patch = {'metadata': {'labels': labels_patch}}

    api = kubernetes.client.CoreV1Api()
    obj = api.patch_namespaced_persistent_volume_claim(
        namespace=namespace,
        name=pvc_name,
        body=pvc_patch,
    )
```

Note that the unrelated labels that were put on the PVC —e.g., manually, from the template, by other controllers/operators, beside the labels coming from the parent EVC— are persisted and never touched (unless the same-named label is applied to EVC and propagated to the PVC).

```
kubectl describe pvc my-claim
```

```
Name:          my-claim
Namespace:     default
StorageClass:  standard
Status:        Bound
Labels:        application=some-app
               owner=me
```

CHAPTER

TEN

CASCADED DELETION

Previously (*Creating the objects* & *Updating the objects* & *Diffing the fields*), we have implemented the creation of a `PersistentVolumeClaim` (PVC) every time an `EphemeralVolumeClaim` (EVC) is created, and cascaded updates of the size and labels when they are changed.

What will happen if the `EphemeralVolumeClaim` is deleted?

```
kubectl delete evc my-claim
kubectl delete -f evc.yaml
```

By default, from the Kubernetes point of view, the PVC & EVC are not connected. Hence, the PVC will continue to exist even if its parent EVC is deleted. Hopefully, some other controller (e.g. the garbage collector) will delete it. Or maybe not.

We want to make sure the child PVC is deleted when the parent EVC is deleted.

The straightforward way would be to implement a deletion handler with `@kopf.on.delete`. But we will go another way, and use the built-in feature of Kubernetes: the owner references.

Let's extend the creation handler:

Listing 1: ephemeral.py

```
import os
import kopf
import kubernetes
import yaml

@kopf.on.create('ephemeralvolumeclaims')
def create_fn(spec, name, namespace, logger, body, **kwargs):

    size = spec.get('size')
    if not size:
        raise kopf.PermanentError(f"Size must be set. Got {size!r}.")

    path = os.path.join(os.path.dirname(__file__), 'pvc.yaml')
    tmpl = open(path, 'rt').read()
    text = tmpl.format(name=name, size=size)
    data = yaml.safe_load(text)

    kopf.adopt(data)

    api = kubernetes.client.CoreV1Api()
    obj = api.create_namespaced_persistent_volume_claim(
```

(continues on next page)

(continued from previous page)

```
        namespace=namespace,
        body=data,
    )

    logger.info(f"PVC child is created: {obj}")

    return {'pvc-name': obj.metadata.name}
```

With this one line, *kopf.adopt()* marks the PVC as a child of EVC. This includes the name auto-generation (if absent), the label propagation, the namespace assignment to the parent's object namespace, and, finally, the owner referencing.

The PVC is now "owned" by the EVC, i.e. it has an owner reference. When the parent EVC object is deleted, the child PVC will also be deleted (and terminated in case of pods), so that we do not need to control this ourselves.

CHAPTER

ELEVEN

CLEANUP

To clean up the cluster after all the experiments are finished:

```
kubectl delete -f obj.yaml
kubectl delete -f crd.yaml
```

Alternatively, Minikube can be reset for the full cluster cleanup.

CHAPTER

TWELVE

HANDLERS

Handlers are Python functions with the actual behaviour of the custom resources.

They are called when any custom resource (within the scope of the operator) is created, modified, or deleted.

Any operator built with Kopf is based on handlers.

12.1 Events & Causes

Kubernetes only notifies when something is changed in the object, but it does not clarify what was changed.

More on that, since Kopf stores the state of the handlers on the object itself, these state changes also cause the events, which are seen by the operators and any other watchers.

To hide the complexity of the state storing, Kopf provides a cause detection: whenever an event happens for the object, the framework detects what happened actually, as follows:

- Was the object just created?
- Was the object deleted (marked for deletion)?
- Was the object edited, and which fields specifically were edited, from what old values into what new values?

These causes, in turn, trigger the appropriate handlers, passing the detected information to the keyword arguments.

12.2 Registering

To register a handler for an event, use the `@kopf.on` decorator:

```
import kopf

@kopf.on.create('kopfexamples')
def my_handler(spec, **_):
    pass
```

All available decorators are described below.

Kopf only supports simple functions and static methods as handlers. Class and instance methods are not supported. For explanation and rationale, see the discussion in #849 (briefly: the semantics of handlers is vague when multiple instances exist or multiple sub-classes inherit from the class, thus inheriting the handlers).

Would you still want to use classes for namespacing, register the handlers by using Kopf's decorators explicitly for specific instances/sub-classes thus resolving the mentioned vagueness and giving the meaning to `self`/`cls`:

```
import kopf

class MyCls:
    def my_handler(self, spec, **kwargs):
        print(repr(self))

instance = MyCls()
kopf.on.create('kopfexamples')(instance.my_handler)
```

12.3 Event-watching handlers

Low-level events can be intercepted and handled silently, without storing the handlers' status (errors, retries, successes) on the object.

This can be useful if the operator needs to watch over the objects of another operator or controller, without adding its data.

The following event-handler is available:

```
import kopf

@kopf.on.event('kopfexamples')
def my_handler(event, **_):
    pass
```

If the event handler fails, the error is logged to the operator's log, and then ignored.

Note: Please note that the event handlers are invoked for *every* event received from the watching stream. This also includes the first-time listing when the operator starts or restarts.

It is the developer's responsibility to make the handlers idempotent (re-executable with no duplicating side-effects).

12.4 State-changing handlers

Kopf goes further and beyond: it detects the actual causes of these events, i.e. what happened to the object:

- Was the object just created?
- Was the object deleted (marked for deletion)?
- Was the object edited, and which fields specifically were edited, from which old values to which new values?

Note: Worth noting that Kopf stores the status of the handlers, such as their progress or errors or retries, in the object itself (`.status` stanza), which triggers its low-level events, but these events are not detected as separate causes, as there is nothing changed *essentially*.

The following 3 core cause-handlers are available:

```
import kopf

@kopf.on.create('kopfexamples')
def my_handler(spec, **_):
    pass

@kopf.on.update('kopfexamples')
def my_handler(spec, old, new, diff, **_):
    pass

@kopf.on.delete('kopfexamples')
def my_handler(spec, **_):
    pass
```

Note: Kopf's finalizers will be added to the object when there are delete handlers specified. Finalizers block Kubernetes from fully deleting objects and Kubernetes will only actually delete objects when all finalizers are removed, i.e. only if the Kopf operator is running to remove them (check: *kubectl freezes on object deletion* for a workaround). If a delete handler is added but finalizers are not required to block the actual deletion, i.e. the handler is optional, the optional argument `optional=True` can be passed to the delete cause decorator.

12.5 Resuming handlers

A special kind of handlers can be used for cases when the operator restarts and detects an object that existed before:

```
@kopf.on.resume('kopfexamples')
def my_handler(spec, **_):
    pass
```

This handler can be used to start threads or asyncio tasks or to update a global state to keep it consistent with the actual state of the cluster. With the resuming handler in addition to creation/update/deletion handlers, no object will be left unattended even if it does not change over time.

The resuming handlers are guaranteed to execute only once per operator lifetime for each resource object (except if errors are retried).

Normally, the resume handlers are mixed-in to the creation and updating handling cycles, and are executed in the order they are declared.

It is a common pattern to declare both creation and resuming handler pointing to the same function, so that this function is called either when an object is created ("started) while the operator is alive ("exists"), or when the operator is started ("created") when the object is existent ("alive"):

```
@kopf.on.resume('kopfexamples')
@kopf.on.create('kopfexamples')
def my_handler(spec, **_):
    pass
```

However, the resuming handlers are **not** called if the object has been deleted during the operator downtime or restart, and the deletion handlers are now being invoked.

This is done intentionally to prevent the cases when the resuming handlers start threads/tasks or allocate the resources, and the deletion handlers stop/free them: it can happen so that the resuming handlers would be executed after the

deletion handlers, thus starting threads/tasks and never stopping them. For example:

```
TASKS = {}

@kopf.on.delete('kopfexamples')
async def my_handler(spec, name, **_):
    if name in TASKS:
        TASKS[name].cancel()

@kopf.on.resume('kopfexamples')
@kopf.on.create('kopfexamples')
def my_handler(spec, **_):
    if name not in TASKS:
        TASKS[name] = asyncio.create_task(some_coroutine(spec))
```

In this example, if the operator starts and notices an object that is marked for deletion, the deletion handler will be called, but the resuming handler is not called at all, despite the object was noticed to exist out there. Otherwise, there would be a resource (e.g. memory) leak.

If the resume handlers are still desired during the deletion handling, they can be explicitly marked as compatible with the deleted state of the object with `deleted=True` option:

```
@kopf.on.resume('kopfexamples', deleted=True)
def my_handler(spec, **_):
    pass
```

In that case, both the deletion and resuming handlers will be invoked. It is the developer's responsibility to ensure this does not lead to memory leaks.

12.6 Field handlers

Specific fields can be handled instead of the whole object:

```
import kopf

@kopf.on.field('kopfexamples', field='spec.somefield')
def somefield_changed(old, new, **_):
    pass
```

There is no special detection of the causes for the fields, such as create/update/delete, so the field-handler is efficient only when the object is updated.

12.7 Sub-handlers

Warning: Sub-handlers are an advanced topic. Please, make sure you understand the regular handlers first, so as the handling cycle of the framework.

A common example for this feature are the lists defined in the spec, each of which should be handled with a handler-like approach rather than explicitly – i.e. with the error tracking, retries, logging, progress and status reporting, etc.

This can be used with dynamically created functions, such as lambdas, partials (`functools.partial`), or the inner functions in the closures:

```
spec:
  items:
    - item1
    - item2
```

Sub-handlers can be implemented either imperatively (where it requires *asynchronous handlers* and `async/await`):

```
import functools
import kopf

@kopf.on.create('kopfexamples')
async def create_fn(spec, **_):
    fns = {}

    for item in spec.get('items', []):
        fns[item] = functools.partial(handle_item, item=item)

   await kopf.execute(fns=fns)

def handle_item(item, *, spec, **_):
    pass
```

Or declaratively with decorators:

```
import kopf

@kopf.on.create('kopfexamples')
def create_fn(spec, **_):

    for item in spec.get('items', []):

        @kopf.subhandler(id=item)
        def handle_item(item=item, **_):
            pass
```

Both of these ways are equivalent. It is a matter of taste and preference which one to use.

The sub-handlers will be processed by all the standard rules and cycles of the Kopf's handling cycle, as if they were the regular handlers with the ids like `create_fn/item1`, `create_fn/item2`, etc.

Warning: The sub-handler functions, their code or their arguments, are not remembered on the object between the handling cycles.

Instead, their parent handler is considered as not finished, and it is called again and again to register the sub-handlers until all the sub-handlers of that parent handler are finished, so that the parent handler also becomes finished.

As such, the parent handler SHOULD NOT produce any side-effects except as the read-only parsing of the inputs (e.g. *spec*), and generating the dynamic functions of the sub-handlers.

CHAPTER

THIRTEEN

DAEMONS

Daemons are a special type of handlers for background logic that accompanies the Kubernetes resources during their life cycle.

Unlike event-driven short-running handlers declared with `@kopf.on`, daemons are started for every individual object when it is created (or when an operator is started/restarted while the object exists), and are capable of running indefinitely (or infinitely) long.

The object's daemons are stopped when the object is deleted or the whole operator is exiting/restarting.

13.1 Spawning

To have a daemon accompanying a resource of some kind, decorate a function with `@kopf.daemon` and make it run for a long time or forever:

```
import asyncio
import time
import kopf

@kopf.daemon('kopfexamples')
async def monitor_kex_async(**kwargs):
    while True:
        ...  # check something
        await asyncio.sleep(10)

@kopf.daemon('kopfexamples')
def monitor_kex_sync(stopped, **kwargs):
    while not stopped:
        ...  # check something
        time.sleep(10)
```

Synchronous functions are executed in threads, asynchronous functions are executed directly in the asyncio event loop of the operator – same as with regular handlers. See *Async/Await*.

The same executor is used both for regular sync handlers and for sync daemons. If you expect a large number of synchronous daemons (e.g. for large clusters), make sure to pre-scale the executor accordingly. See *Configuration* (*Synchronous handlers*).

13.2 Termination

The daemons are terminated when either their resource is marked for deletion, or the operator itself is exiting.

In both cases, the daemons are requested to terminate gracefully by setting the *stopped* kwarg. The synchronous daemons MUST, and asynchronous daemons SHOULD check for the value of this flag as often as possible:

```
import asyncio
import kopf

@kopf.daemon('kopfexamples')
def monitor_kex(stopped, **kwargs):
    while not stopped:
        time.sleep(1.0)
    print("We are done. Bye.")
```

The asynchronous daemons can skip these checks if they define the cancellation timeout. In that case, they can expect an `asyncio.CancelledError` to be raised at any point of their code (specifically, at any `await` clause):

```
import asyncio
import kopf

@kopf.daemon('kopfexamples', cancellation_timeout=1.0)
async def monitor_kex(**kwargs):
    try:
        while True:
            await asyncio.sleep(10)
    except asyncio.CancelledError:
        print("We are done. Bye.")
```

With no cancellation timeout set, cancellation is not performed at all, as it is unclear for how long should the coroutine be awaited. However, it is cancelled when the operator exits and stops all "hung" left-over tasks (not specifically daemons).

Note: The MUST / SHOULD separation is due to Python having no way to terminate a thread unless the thread exits on its own. The *stopped* flag is a way to signal the thread it should exit. If *stopped* is not checked, the synchronous daemons will run forever or until an error happens.

13.3 Timeouts

The termination sequence parameters can be controlled when declaring a daemon:

```
import asyncio
import kopf

@kopf.daemon('kopfexamples',
             cancellation_backoff=1.0, cancellation_timeout=3.0)
async def monitor_kex(stopped, **kwargs):
    while not stopped:
        await asyncio.sleep(1)
```

There are three stages of how the daemon is terminated:

- 1. Graceful termination: * `stopped` is set immediately (unconditionally). * `cancellation_backoff` is awaited (if set).
- 2. Forced termination – only if `cancellation_timeout` is set: * `asyncio.CancelledError` is raised (for async daemons only). * `cancellation_timeout` is awaited (if set).
- 3a. Giving up and abandoning – only if `cancellation_timeout` is set: * A `ResourceWarning` is issued for potential OS resource leaks. * The finalizer is removed, and the object is released for potential deletion.
- 3b. Forever polling – only if `cancellation_timeout` is not set: * The daemon awaiting continues forever, logging from time to time. * The finalizer is not removed and the object remains blocked from deletion.

The `cancellation_timeout` is measured from the point when the daemon is cancelled (forced termination begins), not from where the termination itself begins; i.e., since the moment when the cancellation backoff is over. The total termination time is `cancellation_backoff + cancellation_timeout`.

Warning: When the operator is terminating, it has its timeout of 5 seconds for all "hung" tasks. This includes the daemons after they are requested to finish gracefully and all timeouts are reached.

If the daemon termination takes longer than this for any reason, the daemon will be cancelled (by the operator, not by the daemon guard) regardless of the graceful timeout of the daemon. If this does not help, the operator will be waiting for all hung tasks until SIGKILL'ed.

Warning: If the operator is running in a cluster, there can be timeouts set for a pod (`terminationGracePeriodSeconds`, the default is 30 seconds).

If the daemon termination is longer than this timeout, the daemons will not be finished in full at the operator exit, as the pod will be SIGKILL'ed.

Kopf itself does not set any implicit timeouts for the daemons. Either design the daemons to exit as fast as possible, or configure `terminationGracePeriodSeconds` and cancellation timeouts accordingly.

13.4 Safe sleep

For synchronous daemons, it is recommended to use `stopped.wait()` instead of `time.sleep()`: the wait will end when either the time is reached (as with the sleep), or immediately when the stopped flag is set:

```
import kopf

@kopf.daemon('kopfexamples')
def monitor_kex(stopped, **kwargs):
    while not stopped:
        stopped.wait(10)
```

For asynchronous handlers, regular `asyncio.sleep()` should be sufficient, as it is cancellable via `asyncio.CancelledError`. If a cancellation is neither configured nor desired, `stopped.wait()` can be used too (with `await`):

```
import kopf

@kopf.daemon('kopfexamples')
```

(continues on next page)

(continued from previous page)

```
async def monitor_kex(stopped, **kwargs):
    while not stopped:
        await stopped.wait(10)
```

This way, the daemon will exit as soon as possible when the *stopped* is set, not when the next sleep is over. Therefore, the sleeps can be of any duration while the daemon remains terminable (leads to no OS resource leakage).

Note: Synchronous and asynchronous daemons get different types of stop-checker: with synchronous and asynchronous interfaces respectively. Therefore, they should be used accordingly: without or with `await`.

13.5 Postponing

Normally, daemons are spawned immediately once resource becomes visible to the operator: i.e. on resource creation or operator startup.

It is possible to postpone the daemon spawning:

```
import asyncio
import kopf

@kopf.daemon('kopfexamples', initial_delay=30)
async def monitor_kex(stopped, **kwargs):
    while True:
        await asyncio.sleep(1.0)
```

The start of the daemon will be delayed by 30 seconds after the resource creation (or operator startup). For example, this can be used to give some time for regular event-driven handlers to finish without producing too much activity.

13.6 Restarting

It is generally expected that daemons are designed to run forever. However, a daemon can exit prematurely, i.e. before the resource is deleted or the operator terminates.

In that case, the daemon will not be restarted again during the lifecycle of this resource in this operator process (however, it will be spawned again if the operator restarts). This way, it becomes a long-running equivalent of on-creation/on-resuming handlers.

To simulate restarting, raise *kopf.TemporaryError* with a delay set.

```
import asyncio
import kopf

@kopf.daemon('kopfexamples')
async def monitor_kex(stopped, **kwargs):
    await asyncio.sleep(10.0)
    raise kopf.TemporaryError("Need to restart.", delay=10)
```

Same as with regular error handling, a delay of `None` means instant restart.

See also: *Excluding handlers forever* to prevent daemons from spawning across operator restarts.

13.7 Deletion prevention

Normally, a finalizer is put on the resource if there are daemons running for it – to prevent its actual deletion until all the daemons are terminated.

Only after the daemons are terminated, the finalizer is removed to release the object for actual deletion.

However, it is possible to have daemons that disobey the exiting signals and continue running after the timeouts. In that case, the finalizer is anyway removed, and the orphaned daemons are left to themselves.

13.8 Resource fields access

The resource's current state is accessible at any time through regular kwargs (see *Arguments*): `body`, `spec`, `meta`, `status`, `uid`, `name`, `namespace`, etc.

The values are "live views" of the current state of the object as it is being modified during its lifecycle (not frozen as in the event-driven handlers):

```
import random
import time
import kopf

@kopf.daemon('kopfexamples')
def monitor_kex(stopped, logger, body, spec, **kwargs):
    while not stopped:
        logger.info(f"FIELD={spec['field']}")
        time.sleep(1)

@kopf.timer('kopfexamples', interval=2.5)
def modify_kex_sometimes(patch, **kwargs):
    patch.spec['field'] = random.randint(0, 100)
```

Always access the fields through the provided kwargs, and do not store them in local variables. Internally, Kopf substitutes the whole object's body on every external change. Storing the field values to the variables will remember their value as it was at that moment in time, and will not be updated as the object changes.

13.9 Results delivery

As with any other handlers, the daemons can return arbitrary JSON-serializable values to be put on the resource's status:

```
import asyncio
import kopf

@kopf.daemon('kopfexamples')
async def monitor_kex(stopped, **kwargs):
    await asyncio.sleep(10.0)
    return {'finished': True}
```

13.10 Error handling

The error handling is the same as for all other handlers: see *Error handling*:

```
@kopf.daemon('kopfexamples',
             errors=kopf.ErrorsMode.TEMPORARY, backoff=1, retries=10)
def monitor_kex(retry, **_):
    if retry < 3:
        raise kopf.TemporaryError("I'll be back!", delay=1)
    elif retry < 5:
        raise EnvironmentError("Something happened!")
    else:
        raise kopf.PermanentError("Bye-bye!")
```

If a permanent error is raised, the daemon will never be restarted again. Same as when the daemon exits on its own (but this could be reconsidered in the future).

13.11 Filtering

It is also possible to use the existing *Filtering* to only spawn daemons for specific resources:

```
import time
import kopf

@kopf.daemon('kopfexamples',
             annotations={'some-annotation': 'some-value'},
             labels={'some-label': 'some-value'},
             when=lambda name, **_: 'some' in name)
def monitor_selected_kexes(stopped, **kwargs):
    while not stopped:
        time.sleep(1)
```

Other (non-matching) resources of that kind will be ignored.

The daemons will be executed only while the filtering criteria are met. Both the resource's state and the criteria can be highly dynamic (e.g. due to `when=` callable filters or labels/annotations value callbacks).

Once the daemon stops matching the criteria (either because the resource or the criteria have been changed (e.g. for `when=` callbacks)), the daemon is stopped. Once it matches the criteria again, it is re-spawned.

The checking is done only when the resource changes (any watch-event arrives). The criteria themselves are not re-evaluated if nothing changes.

Warning: A daemon that is terminating is considered as still running, therefore it will not be re-spawned until it fully terminates. It will be re-spawned the next time a watch-event arrives after the daemon has truly exited.

13.12 System resources

Warning: A separate OS thread or asyncio task is started for each resource and each handler.

Having hundreds or thousands of OS threads or asyncio tasks can consume system resources significantly. Make sure you only have daemons and timers with appropriate filters (e.g., by labels, annotations, or so).

For the same reason, prefer to use async handlers (with properly designed async/await code), since asyncio tasks are somewhat cheaper than threads. See *Async/Await* for details.

CHAPTER

FOURTEEN

TIMERS

Timers are schedules of regular handler execution as long as the object exists, no matter if there were any changes or not – unlike the regular handlers, which are event-driven and are triggered only when something changes.

14.1 Intervals

The interval defines how often to trigger the handler (in seconds):

```
import asyncio
import time
import kopf

@kopf.timer('kopfexamples', interval=1.0)
def ping_kex(spec, **kwargs):
    pass
```

14.2 Sharpness

Usually (by default), the timers are invoked with the specified interval between each call. The time taken by the handler itself is not taken into account. It is possible to define timers with a sharp schedule: i.e. invoked every number of seconds sharp, no matter how long it takes to execute it:

```
import asyncio
import time
import kopf

@kopf.timer('kopfexamples', interval=1.0, sharp=True)
def ping_kex(spec, **kwargs):
    time.sleep(0.3)
```

In this example, the timer takes 0.3 seconds to execute. The actual interval between the timers will be 0.7 seconds in the sharp mode: whatever is left of the declared interval of 1.0 seconds minus the execution time.

14.3 Idling

Timers can be defined to idle if the resource changes too often, and only be invoked when it is stable for some time:

```
import asyncio
import kopf

@kopf.timer('kopfexamples', idle=10)
def ping_kex(spec, **kwargs):
    print(f"FIELD={spec['field']}")
```

The creation of a resource is considered as a change, so idling also shifts the very first invocation by that time.

The default is to have no idle time, just the intervals.

It is possible to have a timer with both idling and interval. In that case, the timer will be invoked only if there were no changes in the resource for the specified duration (idle time), and every N seconds after that (interval) as long as the object does not change. Once changed, the timer will stop and wait for the new idling time:

```
import asyncio
import kopf

@kopf.timer('kopfexamples', idle=10, interval=1)
def ping_kex(spec, **kwargs):
    print(f"FIELD={spec['field']}")
```

14.4 Postponing

Normally, timers are invoked immediately once resource becomes visible to the operator (unless idling is declared).

It is possible to postpone the invocations:

```
import asyncio
import time
import kopf

@kopf.timer('kopfexamples', interval=1, initial_delay=5)
def ping_kex(spec, **kwargs):
    print(f"FIELD={spec['field']}")
```

This is similar to idling, except that it is applied only once per resource/operator lifecycle in the very beginning.

14.5 Combined timing

It is possible to combine all scheduled intervals to achieve the desired effect. For example, to give an operator 1 minute for warming up, and then pinging the resources every 10 seconds if they are unmodified for 10 minutes:

```
import kopf

@kopf.timer('kopfexamples',
            initial_delay=60, interval=10, idle=600)
```

(continues on next page)

(continued from previous page)

```
def ping_kex(spec, **kwargs):
    pass
```

14.6 Errors in timers

The timers follow the standard *error handling* protocol: `TemporaryError` and arbitrary exceptions are treated according to the `errors`, `timeout`, `retries`, `backoff` options of the handler. The kwargs *`retry`*, *`started`*, *`runtime`* are provided too.

The default behaviour is to retry arbitrary error (similar to the regular resource handlers).

When an error happens, its delay overrides the timer's schedule or life cycle:

- For arbitrary exceptions, the timer's `backoff=...` option is used.
- For *`kopf.TemporaryError`*, the error's `delay=...` option is used.
- For *`kopf.PermanentError`*, the timer stops forever and is not retried.

The timer's own interval is only used if the function exits successfully.

For example, if the handler fails 3 times with a back-off time set to 5 seconds and the interval set to 10 seconds, it will take 25 seconds (`3*5+10`) from the first execution to the end of the retrying cycle:

```
import kopf

@kopf.timer('kopfexamples',
            errors=kopf.ErrorsMode.TEMPORARY, interval=10, backoff=5)
def monitor_kex_by_time(name, retry, **kwargs):
    if retry < 3:
        raise Exception()
```

It will be executed in that order:

- A new cycle begins: * 1st execution attempt fails (`retry == 0`). * Waits for 5 seconds (`backoff`). * 2nd execution attempt fails (`retry == 1`). * Waits for 5 seconds (`backoff`). * 3rd execution attempt fails (`retry == 2`). * Waits for 5 seconds (`backoff`). * 4th execution attempt succeeds (`retry == 3`). * Waits for 10 seconds (`interval`).
- A new cycle begins: * 5th execution attempt fails (`retry == 0`).

The timer never overlaps with itself. Though, multiple timers with different interval settings and execution schedules can eventually overlap with each other and with event-driven handlers.

14.7 Results delivery

The timers follow the standard *results delivery* protocol: the returned values are put on the object's status under the handler's id as a key.

```
import random
import kopf

@kopf.timer('kopfexamples', interval=10)
```

(continues on next page)

(continued from previous page)

```
def ping_kex(spec, **kwargs):
    return random.randint(0, 100)
```

Note: Whenever a resulting value is serialised and put on the resource's status, it modifies the resource, which, in turn, resets the idle timer. Use carefully with both idling & returned results.

14.8 Filtering

It is also possible to use the existing *Filtering*:

```
import kopf

@kopf.timer('kopfexamples', interval=10,
            annotations={'some-annotation': 'some-value'},
            labels={'some-label': 'some-value'},
            when=lambda name, **_: 'some' in name)
def ping_kex(spec, **kwargs):
    pass
```

14.9 System resources

Warning: Timers are implemented the same way as asynchronous daemons (see *Daemons*) — via asyncio tasks for every resource & handler.

Despite OS threads are not involved until the synchronous functions are invoked (through the asyncio executors), this can lead to significant OS resource usage on large clusters with thousands of resources.

Make sure you only have daemons and timers with appropriate filters (e.g., by labels, annotations, or so).

CHAPTER

FIFTEEN

ARGUMENTS

15.1 Forward compatibility kwargs

`**kwargs` is required in all handlers for the forward compatibility: the framework can add new keywords in the future, and the existing handlers should accept them without breaking, even if they do not use them.

It can be named `**_` to prevent the "unused variable" warnings by linters.

15.2 Retrying and timing

Most (but not all) of the handlers – such as resource change detection, resource daemons and timers, and activity handlers – are capable of retrying their execution in case of errors (see also: *Error handling*). They provide kwargs regarding the retrying process:

`retry` (`int`) is the sequential number of retry of this handler. For the first attempt, it is `0`, so it can be used in expressions like `if not retry: ...`.

`started` (`datetime.datetime`) is the start time of the handler, in case of retries & errors – i.e. of the first attempt.

`runtime` (`datetime.timedelta`) is the duration of the handler run, in case of retries & errors – i.e. since the first attempt.

15.3 Parametrization

`param` (any type, defaults to `None`) is a value passed from the same-named handler option `param=`. It can be helpful if there are multiple decorators, possibly with multiple different selectors & filters, for one handler function:

```
import kopf

@kopf.on.create('KopfExample', param=1000)
@kopf.on.resume('KopfExample', param=100)
@kopf.on.update('KopfExample', param=10, field='spec.field')
@kopf.on.update('KopfExample', param=1, field='spec.items')
def count_updates(param, patch, **_):
    patch.status['counter'] = body.status.get('counter', 0) + param

@kopf.on.update('Child1', param='first', field='status.done', new=True)
@kopf.on.update('Child2', param='second', field='status.done', new=True)
def child_updated(param, patch, **_):
    patch_parent({'status': {param: {'done': True}}})
```

Note that Kopf deduplicates the handlers to execute on one single occasion by their underlying function and its id, which includes the field name by default.

In this example below with overlapping criteria, if `spec.field` is updated, the handler will be called twice: one time – for `spec` as a whole, another time – for `spec.field` in particular; each time with the proper values of old/new/diff/param kwargs for those fields:

```
import kopf

@kopf.on.update('KopfExample', param=10, field='spec.field')
@kopf.on.update('KopfExample', param=1, field='spec')
def fn(param, **_):
    pass
```

15.4 Operator configuration

`settings` is passed to activity handlers (but not to resource handlers).

It is an object with a predefined nested structure of containers with values, which defines the operator's behaviour. See: *`kopf.OperatorSettings`*.

It can be modified if needed (usually in the startup handlers). Every operator (if there are more than one in the same process) has its config.

See also: *Configuration*.

15.5 Resource-related kwargs

15.5.1 Body parts

`resource` (*`kopf.Resource`*) is the actual resource being served as retrieved from the cluster during the initial discovery. Please note that it is not necessary the same selector as used in the decorator, as one selector can match multiple actual resources.

`body` is the handled object's body, a read-only mapping (dict). It might look like this as an example:

```
{
    'apiVersion': 'kopf.dev/v1',
    'kind': 'KopfExample',
    'metadata': {
        'name': 'kopf-example-1',
        'namespace': 'default',
        'uid': '1234-5678-...',
    },
    'spec': {
        'field': 'value',
    },
    'status': {
        ...
    },
}
```

`spec`, `meta`, `status` are aliases for relevant stanzas, and are live-views into `body['spec']`, `body['metadata']`, `body['status']`.

`namespace`, `name`, `uid` can be used to identify the object being handled, and are aliases for the respective fields in `body['metadata']`. If the values are not present for any reason (e.g. namespaced for cluster-scoped objects), the fields are `None` – unlike accessing the same fields by key, when a `KeyError` is raised.

`labels` and `annotations` are equivalents of `body['metadata']['labels']` and `body['metadata']['annotations']` if they exist. If not, these two behave as empty dicts.

15.5.2 Logging

`logger` is a per-object logger, with the messages prefixed with the object's namespace/name.

Some of the log messages are also sent as Kubernetes events according to the log-level configuration (default is INFO, WARNINGs, ERRORs).

15.5.3 Patching

`patch` is a mutable mapping (dict) with the object changes to be applied after the handler. It is actively used internally by the framework itself, and is shared to the handlers for convenience _(since patching happens anyway in the framework, why make separate API calls for patching?)_.

Note: Currently, it is just a dictionary, and the changes are applied as `application/merge-patch+json`: `None` values delete the fields, other values override, dicts are merged.

In the future, at discretion of this framework, it can be converted to JSON-patch (a list of add/change/remove operation), while keeping the same Python mutable mapping protocol and remembering the changes in the order they were made.

15.5.4 In-memory container

`memo` is an in-memory container for arbitrary runtime-only keys-values. The values can be accessed as either object attributes or dictionary keys.

For resource handlers, `memo` is shared by all handlers of the same individual resource (not of the resource kind, but of the resource object). For operator handlers, `memo` is shared by all handlers of the same operator and later used to populate the resources' `memo` containers.

See also:

In-memory containers and *`kopf.Memo`*.

15.5.5 In-memory indices

Indices are in-memory overviews of matching resources in the cluster. They are populated according to `@kopf.index` handlers and their filters.

Each index is exposed in kwargs under its name (function name) or id (if overridden with `id=`). There is no global structure to access all indices at once. If needed, use `**kwargs` itself.

Indices are available for all operator-level and all resource-level handlers. For resource handlers, they are guaranteed to be populated before any handlers are invoked. For operator handlers, there is no such guarantee.

See also:

In-memory indexing.

15.6 Resource-watching kwargs

For the resource watching handlers, an extra kwarg is provided:

15.6.1 API event

`event` is a raw JSON-decoded message received from Kubernetes API; it is a dict with `['type']` & `['object']` keys.

15.7 Resource-changing kwargs

Kopf provides functionality for change detection and triggers the handlers for those changes (not for every event coming from the Kubernetes API). A few extra kwargs are provided for these handlers, exposing the changes:

15.7.1 Causation

`reason` is a type of change detection (creation, update, deletion, resuming). It is generally reflected in the handler decorator used, but can be useful for the multi-purpose handlers pointing to the same function (e.g. for `@kopf.on.create` + `@kopf.on.resume` pairs).

15.7.2 Diffing

`old` & `new` are the old & new state of the object or a field within the detected changes. The new state usually corresponds to *body*.

For the whole-object handlers, `new` is an equivalent of *body*. For the field handlers, it is the value of that field specifically.

`diff` is a list of changes of the object between old & new states.

The diff highlights which keys were added, changed, or removed in the dictionary, with old & new values being selectable, and generally ignores all other fields that were not changed.

Due to specifics of Kubernetes, `None` is interpreted as absence of the value/field, not as a value of its own kind. In case of diffs, it means that the value did not exist before, or will not exist after the changes (for the old & new value positions respectively):

15.8 Resource daemon kwargs

15.8.1 Stop-flag

Daemons also have `stopped`. It is a flag object for sync & async daemons (mostly, sync) to check if they should stop. See also: `DaemonStopped`.

To check, `.is_set()` method can be called, or the object itself can be used as a boolean expression: e.g. `while not stopped: ...`.

Its `.wait()` method can be used to replace `time.sleep()` or `asyncio.sleep()` for faster (instant) termination on resource deletion.

See more: *Daemons*.

15.9 Resource admission kwargs

15.9.1 Dry run

Admission handlers, both validating and mutating, must skip any side effects if `dryrun` is `True`. It is `True` when a dry-run API request is made, e.g. with `kubectl --dry-run=server ...`.

Regardless of `dryrun`, the handlers must not make any side effects unless they declare themselves as `side_effects=True`.

See more: *Admission control*.

15.9.2 Subresources

`subresource` (`str|None`) is the name of a subresource being checked. `None` means that the main body of the resource is being checked. Otherwise, it is usually `"status"` or `"scale"`; other values are possible. (The value is never `"*"`, as the star mask is used only for handler filters.)

See more: *Admission control*.

15.9.3 Admission warnings

`warnings` (`list[str]`) is a **mutable** list of string used as warnings. The admission webhook handlers can populate the list with warnings (strings), and the webhook servers/tunnels return them to Kubernetes, which shows them to `kubectl`.

See more: *Admission control*.

15.9.4 User information

`userinfo` (`Mapping[str, Any]`) is an information about a user that sends the API request to Kubernetes.

It usually contains the keys `'username'`, `'uid'`, `'groups'`, but this might change in the future. The information is provided exactly as Kubernetes sends it in the admission request.

See more: *Admission control*.

15.9.5 Request credentials

For rudimentary authentication and authorization, Kopf passes the information from the admission requests to the admission handlers as is, without additional interpretation of it.

`headers` (`Mapping[str, str]`) contains all HTTPS request headers, including `Authorization: Basic ...`, `Authorization: Bearer ...`.

`sslpeer` (`Mapping[str, Any]`) contains the SSL peer information as returned by `ssl.SSLSocket.getpeercert()`. It is `None` if no proper SSL client certificate was provided (i.e. by apiservers talking to webhooks), or if the SSL protocol could not verify the provided certificate with its CA.

Note: This is an identity of the apiservers that send the admission request, not of the user or an app that sends the API request to Kubernetes. For the user's identity, use *`userinfo`*.

See more: *Admission control*.

CHAPTER

SIXTEEN

ASYNC/AWAIT

Kopf supports asynchronous handler functions:

```
import asyncio
import kopf

@kopf.on.create('kopfexamples')
async def create_fn(spec, **_):
    await asyncio.sleep(1.0)
```

Async functions have an additional benefit over the non-async ones to make the full stack trace available when exceptions occur or IDE breakpoints are used since the async functions are executed directly inside of Kopf's event loop in the main thread.

Regular synchronous handlers, despite supported for convenience, are executed in parallel threads (via the default executor of the loop), and can only see the stack traces up to the thread entry point.

Warning: As with any async coroutines, it is the developer's responsibility to make sure that all the internal function calls are either `await`s of other async coroutines (e.g. `await asyncio.sleep()`), or the regular non-blocking functions calls.

Calling a synchronous function (e.g. HTTP API calls or `time.sleep()`) inside of an asynchronous function will block the whole operator process until the synchronous call if finished, i.e. even other resources processed in parallel, and the Kubernetes event-watching/-queueing cycles.

This can come unnoticed in the development environment with only a few resources and no external timeouts, but can hit hard in the production environments with high load.

CHAPTER

SEVENTEEN

LOADING AND IMPORTING

Kopf requires the source files with the handlers on the command line. It does not do any attempts to guess the user's intentions or to introduce any conventions (at least, now).

There are two ways to specify them (both mimicking the Python interpreter):

- Direct script files:

```
kopf run file1.py file2.py
```

- Importable modules:

```
kopf run -m package1.module1 -m package2.module2
```

- Or mixed:

```
kopf run file1.py file2.py -m package1.module1 -m package2.module2
```

Which way to use depends on how the source code is structured, and is out of the scope of Kopf.

Each of the mentioned files and modules will be imported. The handlers should be registered during the import. This is usually done by using the function decorators — see *Handlers*.

CHAPTER

EIGHTEEN

RESOURCE SPECIFICATION

The following notations are supported to specify the resources to be handled. As a rule of thumb, they are designed so that the intentions of a developer are guessed the best way possible, and similar to `kubectl` semantics.

The resource name is always expected in the first place as the rightmost value. The remaining parts are considered as an API group and an API version of the resource – given as either two separate strings, or as one, but separated with a slash:

```
@kopf.on.event('kopf.dev', 'v1', 'kopfexamples')
@kopf.on.event('kopf.dev/v1', 'kopfexamples')
@kopf.on.event('apps', 'v1', 'deployments')
@kopf.on.event('apps/v1', 'deployments')
@kopf.on.event('', 'v1', 'pods')
def fn(**_):
    pass
```

If only one API specification is given (except for `v1`), it is treated as an API group, and the preferred API version of that API group is used:

```
@kopf.on.event('kopf.dev', 'kopfexamples')
@kopf.on.event('apps', 'deployments')
def fn(**_):
    pass
```

It is also possible to specify the resources with `kubectl`'s semantics:

```
@kopf.on.event('kopfexamples.kopf.dev')
@kopf.on.event('deployments.apps')
def fn(**_):
    pass
```

One exceptional case is `v1` as the API specification: it corresponds to K8s's legacy core API (before API groups appeared), and is equivalent to an empty API group name. The following specifications are equivalent:

```
@kopf.on.event('v1', 'pods')
@kopf.on.event('', 'v1', 'pods')
def fn(**_):
    pass
```

If neither the API group nor the API version is specified, all resources with that name would match regardless of the API groups/versions. However, it is reasonable to expect only one:

```
@kopf.on.event('kopfexamples')
@kopf.on.event('deployments')
@kopf.on.event('pods')
def fn(**_):
    pass
```

In all examples above, where the resource identifier is expected, it can be any name: plural, singular, kind, or a short name. As it is impossible to guess which one is which, the name is remembered as is, and is later checked for all possible names of the specific resources once those are discovered:

```
@kopf.on.event('kopfexamples')
@kopf.on.event('kopfexample')
@kopf.on.event('KopfExample')
@kopf.on.event('kex')
@kopf.on.event('StatefulSet')
@kopf.on.event('deployments')
@kopf.on.event('pod')
def fn(**_):
    pass
```

The resource specification can be more specific on which name to match:

```
@kopf.on.event(kind='KopfExample')
@kopf.on.event(plural='kopfexamples')
@kopf.on.event(singular='kopfexample')
@kopf.on.event(shortcut='kex')
def fn(**_):
    pass
```

The whole categories of resources can be served, but they must be explicitly specified to avoid unintended consequences:

```
@kopf.on.event(category='all')
def fn(**_):
    pass
```

Note that the conventional category `all` does not really mean all resources, but only those explicitly added to this category; some built-in resources are excluded (e.g. ingresses, secrets).

To handle all resources in an API group/version, use a special marker instead of the mandatory resource name:

```
@kopf.on.event('kopf.dev', 'v1', kopf.EVERYTHING)
@kopf.on.event('kopf.dev/v1', kopf.EVERYTHING)
@kopf.on.event('kopf.dev', kopf.EVERYTHING)
def fn(**_):
    pass
```

As a consequence of the above, to handle every resource in the cluster – which might be not the best idea per se, but is technically possible – omit the API group/version, and use the marker only:

```
@kopf.on.event(kopf.EVERYTHING)
def fn(**_):
    pass
```

Serving everything is better when it is used with filters:

```
@kopf.on.event(kopf.EVERYTHING, labels={'only-this': kopf.PRESENT})
def fn(**_):
    pass
```

Note: Core v1 events are excluded from `EVERYTHING`: they are created during handling of other resources in the implicit *Events* from log messages, so they would cause unnecessary handling cycles for every essential change.

To handle core v1 events, they must be named explicitly, e.g. like this:

```
@kopf.on.event('v1', 'events')
def fn(**_):
    pass
```

The resource specifications do not support multiple values, masks or globs. To handle multiple independent resources, add multiple decorators to the same handler function – as shown above. The handlers are deduplicated by the underlying function and its handler id (which, in turn, equals to the function's name by default unless overridden), so one function will never be triggered multiple times for the same resource if there are some accidental overlaps in the specifications.

Warning: Kopf tries to make it easy to specify resources a la `kubectl`. However, some things cannot be made that easy. If resources are specified ambiguously, i.e. if 2+ resources of different API groups match the same resource specification, neither of them will be served, and a warning will be issued.

This only applies to resource specifications where it is intended to have a specific resource by its name; specifications with intentional multi-resource mode are served as usually (e.g. by categories).

However, `v1` resources have priority over all other resources. This resolves the conflict of `pods.v1` vs. `pods.v1beta1.metrics.k8s.io`, so just `"pods"` can be specified and the intention will be understood.

This mimics the behaviour of `kubectl`, where such API priorities are hard-coded.

While it might be convenient to write short forms of resource names, the proper way is to always add at least an API group:

```
import kopf

@kopf.on.event('pods')  # NOT SO GOOD, ambiguous, though works
@kopf.on.event('pods.v1')  # GOOD, specific
@kopf.on.event('v1', 'pods')  # GOOD, specific
@kopf.on.event('pods.metrics.k8s.io')  # GOOD, specific
@kopf.on.event('metrics.k8s.io', 'pods')  # GOOD, specific
def fn(**_):
    pass
```

Keep the short forms only for prototyping and experimentation mode, and for ad-hoc operators with custom resources (not reusable and running in controlled clusters where no other similar resources can be defined).

Warning: Some API groups are served by API extensions: e.g. `metrics.k8s.io`. If the extension's deployment/service/pods are down, such a group will not be scannable (failing with "HTTP 503 Service Unavailable") and will block scanning the whole cluster if resources are specified with no group name (e.g. `('pods')` instead of `('v1', 'pods')`).

To avoid scanning the whole cluster and all (even unused) API groups, it is recommended to specify at least the

group names for all resources, especially in reusable and publicly distributed operators.

CHAPTER

NINETEEN

FILTERING

Handlers can be restricted to only the resources that match certain criteria.

Multiple criteria are joined with AND, i.e. they all must be satisfied.

Unless stated otherwise, the described filters are available for all handlers: resuming, creation, deletion, updating, event-watching, timers, daemons, or even to sub-handlers (thus eliminating some checks in its parent's code).

There are only a few kinds of checks:

- Specific values – expressed with Python literals such as `"a string"`.
- Presence of values – with special markers `kopf.PRESENT`/`kopf.ABSENT`.
- Per-value callbacks – with anything callable which evaluates to true/false.
- Whole-body callbacks – with anything callable which evaluates to true/false.

But there are multiple places where these checks can be applied, each has its specifics.

19.1 Metadata filters

Metadata is the most commonly filtered aspect of the resources.

Match only when the resource's label or annotation has a specific value:

```
@kopf.on.create('kopfexamples',
                labels={'some-label': 'somevalue'},
                annotations={'some-annotation': 'somevalue'})
def my_handler(spec, **_):
    pass
```

Match only when the resource has a label or an annotation with any value:

```
@kopf.on.create('kopfexamples',
                labels={'some-label': kopf.PRESENT},
                annotations={'some-annotation': kopf.PRESENT})
def my_handler(spec, **_):
    pass
```

Match only when the resource has no label or annotation with that name:

```
@kopf.on.create('kopfexamples',
                labels={'some-label': kopf.ABSENT},
                annotations={'some-annotation': kopf.ABSENT})
```

(continues on next page)

(continued from previous page)

```
def my_handler(spec, **_):
    pass
```

Note that empty strings in labels and annotations are treated as regular values, i.e. they are considered as present on the resource.

19.2 Field filters

Specific fields can be checked for specific values or presence/absence, similar to the metadata filters:

```
@kopf.on.create('kopfexamples', field='spec.field', value='world')
def created_with_world_in_field(**_):
    pass

@kopf.on.create('kopfexamples', field='spec.field', value=kopf.PRESENT)
def created_with_field(**_):
    pass

@kopf.on.create('kopfexamples', field='spec.no-field', value=kopf.ABSENT)
def created_without_field(**_):
    pass
```

When the `value=` filter is not specified, but the `field=` filter is, it is equivalent to `value=kopf.PRESENT`, i.e. the field must be present with any value (for update handlers: present before or after the change).

```
@kopf.on.create('kopfexamples', field='spec.field')
def created_with_field(**_):
    pass

@kopf.on.update('kopfexamples', field='spec.field')
def field_is_affected(old, new, **_):
    pass
```

Since the field name is part of the handler id (e.g., `"fn/spec.field"`), multiple decorators can be defined to react to different fields with the same function and it will be invoked multiple times with different old & new values relevant to the specified fields, so as different values of *`param`*:

```
@kopf.on.update('kopfexamples', field='spec.field', param='fld')
@kopf.on.update('kopfexamples', field='spec.items', param='itm')
def one_of_the_fields_is_affected(old, new, **_):
    pass
```

However, different causes –mostly resuming + one of creation/update/deletion– will not be distinguished, so e.g. resume+create pair with the same field will be called only once.

Due to the special nature of update handlers (`@on.update`, `@on.field`), described in a note below, this filtering semantics is extended for them:

The `field=` filter restricts the update-handlers to cases when the specified field is in any way affected: changed, added or removed to/from the resource. When the specified field is not affected, but something else is changed, such update-handlers are not invoked even if they do match the field criteria.

The `value=` filter applies to either the old or the new value: i.e. if any of them satisfies the value criterion. This covers both sides of the state transition: when the value criterion has just been satisfied (though was not satisfied before), or when the value criterion was satisfied before (but stopped being satisfied). For the latter case, it means that the transitioning resource still satisfies the filter in its "old" state.

Note: **Technically,** the update handlers are called after the change has already happened on the low level – i.e. when the field already has the new value.

Semantically, the update handlers are only initiated by this change, but are executed before the current (new) state is processed and persisted, thus marking the end of the change processing cycle – i.e. they are called in-between the old and new states, and therefore belong to both of them.

In general, the resource-changing handlers are an abstraction on top of the low-level K8s machinery for eventual processing of such state transitions, so their semantics can differ from K8s's low-level semantics. In most cases, this is not visible or important to the operator developers, except for such cases, where it might affect the semantics of e.g. filters.

For reacting to *unrelated* changes of other fields while this field satisfies the criterion, use `when=` instead of `field=/value=`.

For reacting to only the cases when the desired state is reached but not when the desired state is lost, use `new=` with the same criterion; similarly, for the cases when the desired state is only lost, use `old=`.

For all other handlers with no concept of "updating" and being in-between of two equally valid and applicable states, the `field=/value=` filters check the resource in its current –and the only– state. The handlers are being invoked and the daemons are running as long as the field and the value match the criterion.

19.3 Change filters

The update handlers (specifically, `@kopf.on.update` and `@kopf.on.field`) check the `value=` filter against both old & new values, which might be not what is intended. For more precision on filtering, the old/new values can be checked separately with the `old=/new=` filters with the same filtering methods/markers as all other filters.

```
@kopf.on.update('kopfexamples', field='spec.field', old='x', new='y')
def field_is_edited(**_):
    pass

@kopf.on.update('kopfexamples', field='spec.field', old=kopf.ABSENT, new=kopf.PRESENT)
def field_is_added(**_):
    pass

@kopf.on.update('kopfexamples', field='spec.field', old=kopf.PRESENT, new=kopf.ABSENT)
def field_is_removed(**_):
    pass
```

If one of `old=` or `new=` is not specified (or set to `None`), that part is not checked, but the other (specified) part is still checked:

Match when the field reaches a specific value either by being edited/patched to it or by adding it to the resource (i.e. regardless of the old value):

```
@kopf.on.update('kopfexamples', field='spec.field', new='world')
def hello_world(**_):
    pass
```

Match when the field loses a specific value either by being edited/patched to something else, or by removing the field from the resource:

```
@kopf.on.update('kopfexamples', field='spec.field', old='world')
def goodbye_world(**_):
    pass
```

Generally, the update handlers with `old=`/`new=` filters are invoked only when the field's value is changed, and are not invoked when it remains the same.

For clarity, "a change" means not only an actual change of the value, but also a change in the field's presence or absence in the resource.

If none of the `old=`/`new=`/`value=` filters is specified, the handler is invoked if the field is affected in any way, i.e. if it was modified, added, or removed. This is the same behaviour as with the unspecified `value=` filter.

Note: `value=` is currently made to be mutually exclusive with `old=`/`new=`: only one filtering method can be used; if both methods are used together, it would be ambiguous. This can be reconsidered in the future.

19.4 Value callbacks

Instead of specific values or special markers, all the value-based filters can use arbitrary per-value callbacks (as an advanced use-case for advanced logic).

The value callbacks must receive the same *keyword arguments* as the respective handlers (with `**kwargs`/`**_` for forwards compatibility), plus one *positional* (not keyword!) argument with the value being checked. The passed value will be `None` if the value is absent in the resource.

```
def check_value(value, spec, **_):
    return value == 'some-value' and spec.get('field') is not None

@kopf.on.create('kopfexamples',
                labels={'some-label': check_value},
                annotations={'some-annotation': check_value})
def my_handler(spec, **_):
    pass
```

19.5 Callback filters

The resource callbacks must receive the same *keyword arguments* as the respective handlers (with `**kwargs`/`**_` for forwards compatibility).

```
def is_good_enough(spec, **_):
    return spec.get('field') in spec.get('items', [])

@kopf.on.create('kopfexamples', when=is_good_enough)
def my_handler(spec, **_):
    pass

@kopf.on.create('kopfexamples', when=lambda spec, **_: spec.get('field') in spec.get(
```

(continues on next page)

(continued from previous page)

```
↪'items', []))
def my_handler(spec, **_):
    pass
```

There is no need for the callback filters to only check the resource's content. They can filter by any kwarg data, e.g. by a *reason* of this invocation, remembered *memo* values, etc. However, it is highly recommended that the filters do not modify the state of the operator – keep it for handlers.

19.6 Callback helpers

Kopf provides several helpers to combine multiple callbacks into one (the semantics is the same as for Python's built-in functions):

```
import kopf

def whole_fn1(name, **_): return name.startswith('kopf-')
def whole_fn2(spec, **_): return spec.get('field') == 'value'
def value_fn1(value, **_): return value.startswith('some')
def value_fn2(value, **_): return value.endswith('label')

@kopf.on.create('kopfexamples',
                when=kopf.all_([whole_fn1, whole_fn2]),
                labels={'somelabel': kopf.all_([value_fn1, value_fn2])})
def create_fn1(**_):
    pass

@kopf.on.create('kopfexamples',
                when=kopf.any_([whole_fn1, whole_fn2]),
                labels={'somelabel': kopf.any_([value_fn1, value_fn2])})
def create_fn2(**_):
    pass
```

The following wrappers are available:

- `kopf.not_(fn)` – the function must return `False` to pass the filters.
- `kopf.any_([...])` – at least one of the functions must return `True`.
- `kopf.all_([...])` – all of the functions must return `True`.
- `kopf.none_([...])` – all of the functions must return `False`.

19.7 Stealth mode

Note: Please note that if an object does not match any filters of any handlers for its resource kind, there will be no messages logged and no annotations stored on the object. Such objects are processed in the stealth mode even if the operator technically sees them in the watch-stream.

As the result, when the object is updated to match the filters some time later (e.g. by putting labels/annotations on it, or changing its spec), this will not be considered as an update but as a creation.

From the operator's point of view, the object has suddenly appeared in sight with no diff-base, which means that it is a newly created object; so, the on-creation handlers will be called instead of the on-update ones.

This behaviour is correct and reasonable from the filtering logic side. If this is a problem, then create a dummy handler without filters (e.g. a field-handler for a non-existent field) – this will make all the objects always being in the scope of the operator, even if the operator did not react to their creation/update/deletion, and so the diff-base annotations ("last-handled-configuration", etc) will be always added on the actual object creation, not on scope changes.

CHAPTER

TWENTY

RESULTS DELIVERY

All handlers can return arbitrary JSON-serializable values. These values are then put to the resource status under the name of the handler:

```
import kopf

@kopf.on.create('kopfexamples')
def create_kex_1(**_):
    return 100

@kopf.on.create('kopfexamples')
def create_kex_2(uid, **_):
    return {'r1': random.randint(0, 100), 'r2': random.randint(100, 999)}
```

These results can be seen in the object's content:

```
$ kubectl get -o yaml kex kopf-example-1
```

```
...
status:
  create_kex_1: 100
  create_kex_2:
    r1: 66
    r2: 666
```

The function results can be used to communicate between handlers through resource itself, assuming that handlers do not know in which order they will be invoked (due to error handling and retrying), and to be able to restore in case of operator failures & restarts:

```
import kopf
import pykube

@kopf.on.create('kopfexamples')
def create_job(status, **_):
    if not status.get('create_pvc', {}):
        raise kopf.TemporaryError("PVC is not created yet.", delay=10)

    pvc_name = status['create_pvc']['name']

    api = pykube.HTTPClient(pykube.KubeConfig.from_env())
    obj = pykube.Job(api, {...})  # use pvc_name here
    obj.create()
```

(continues on next page)

(continued from previous page)

```
    return {'name': obj.name}

@kopf.on.create('kopfexamples')
def create_pvc(**_):
    api = pykube.HTTPClient(pykube.KubeConfig.from_env())
    obj = pykube.PersistentVolumeClaim(api, {...})
    obj.create()
    return {'name': obj.name}
```

Note: In this example, the handlers are *intentionally* put in such an order that the first handler always fails on the first attempt. Having them in the proper order (PVC first, Job afterwards) will make it work smoothly for most of the cases, until PVC creation fails for any temporary reason and has to be retried. The whole thing will eventually succeed anyway in 1-2 additional retries, just with less friendly messages and stack traces.

CHAPTER

TWENTYONE

ERROR HANDLING

Kopf tracks the status of the handlers (except for the low-level event handlers) catches the exceptions, and processes them from each of the handlers.

The last (or the final) exception is stored in the object's status, and reported via the object's events.

Note: Keep in mind, the Kubernetes events are often garbage-collected fast, e.g. less than 1 hour, so they are visible only soon after they are added. For persistence, the errors are also stored on the object's status.

21.1 Temporary errors

If an exception raised inherits from *kopf.TemporaryError*, it will postpone the current handler for the next iteration, which can happen either immediately, or after some delay:

```
import kopf

@kopf.on.create('kopfexamples')
def create_fn(spec, **_):
    if not is_data_ready():
        raise kopf.TemporaryError("The data is not yet ready.", delay=60)
```

In that case, there is no need to sleep in the handler explicitly, thus blocking any other events, causes, and generally any other handlers on the same object from being handled (such as deletion or parallel handlers/sub-handlers).

Note: The multiple handlers and the sub-handlers are implemented via this kind of errors: if there are handlers left after the current cycle, a special retriable error is raised, which marks the current cycle as to be retried immediately, where it continues with the remaining handlers.

The only difference is that this special case produces fewer logs.

21.2 Permanent errors

If a raised exception inherits from *kopf.PermanentError*, the handler is considered as non-retriable and non-recoverable and completely failed.

Use this when the domain logic of the application means that there is no need to retry over time, as it will not become better:

```
import kopf

@kopf.on.create('kopfexamples')
def create_fn(spec, **_):
    valid_until = datetime.datetime.fromisoformat(spec['validUntil'])
    if valid_until <= datetime.datetime.now(datetime.timezone.utc):
        raise kopf.PermanentError("The object is not valid anymore.")
```

See also: *Excluding handlers forever* to prevent handlers from being invoked for the future change-sets even after the operator restarts.

21.3 Regular errors

Kopf assumes that any arbitrary errors (i.e. not *kopf.TemporaryError* and not *kopf.PermanentError*) are the environment's issues and can self-resolve after some time.

As such, as default behaviour, Kopf retries the handlers with arbitrary errors infinitely until the handlers either succeed or fail permanently.

The reaction to the arbitrary errors can be configured:

```
import kopf

@kopf.on.create('kopfexamples', errors=kopf.ErrorsMode.PERMANENT)
def create_fn(spec, **_):
    raise Exception()
```

Possible values of `errors` are:

- `kopf.ErrorsMode.TEMPORARY` (the default).
- `kopf.ErrorsMode.PERMANENT` (prevent retries).
- `kopf.ErrorsMode.IGNORED` (same as in the resource watching handlers).

21.4 Timeouts

The overall runtime of the handler can be limited:

```
import kopf

@kopf.on.create('kopfexamples', timeout=60*60)
def create_fn(spec, **_):
    raise kopf.TemporaryError(delay=60)
```

If the handler is not succeeded within this time, it is considered as fatally failed.

If the handler is an async coroutine and it is still running at the moment, an `asyncio.TimeoutError` is raised; there is no equivalent way of terminating the synchronous functions by force.

By default, there is no timeout, so the retries continue forever.

21.5 Retries

The number of retries can be limited too:

```
import kopf

@kopf.on.create('kopfexamples', retries=3)
def create_fn(spec, **_):
    raise Exception()
```

Once the number of retries is reached, the handler fails permanently.

By default, there is no limit, so the retries continue forever.

21.6 Backoff

The interval between retries on arbitrary errors, when an external environment is supposed to recover and be able to succeed the handler execution, can be configured:

```
import kopf

@kopf.on.create('kopfexamples', backoff=30)
def create_fn(spec, **_):
    raise Exception()
```

The default is 60 seconds.

Note: This only affects the arbitrary errors. When `TemporaryError` is explicitly used, the delay should be configured with `delay=...`.

CHAPTER

TWENTYTWO

SCOPES

22.1 Namespaces

An operator can be restricted to handle custom resources in one namespace only:

```
kopf run --namespace=some-namespace ...
kopf run -n some-namespace ...
```

Multiple namespaces can be served:

```
kopf run --namespace=some-namespace --namespace=another-namespace ...
kopf run -n some-namespace -n another-namespace ...
```

Namespace globs with * and ? characters can be used too:

```
kopf run --namespace=*-pr-123-* ...
kopf run -n *-pr-123-* ...
```

Namespaces can be negated: all namespaces are served except those excluded:

```
kopf run --namespace=!*-pr-123-* ...
kopf run -n !*-pr-123-* ...
```

Multiple globs can be used in one pattern. The rightmost matching one wins. The first glob is decisive: if a namespace does not match it, it does not match the whole pattern regardless of what is there (other globs are not checked). If the first glob is a negation, it is implied that initially, all namespaces do match (as if preceded by *), and then the negated ones are excluded.

In this artificial example, `myapp-live` will match, `myapp-pr-456` will not match, but `myapp-pr-123` will match; `otherapp-live` will not match; even `otherapp-pr-123` will not match despite the `-pr-123` suffix in it because it does not match the initial decisive glob:

```
kopf run --namespace=myapp-*,!*-pr-*,*-pr-123 ...
```

In all cases, the operator monitors the namespaces that exist at the startup or are created/deleted at runtime, and starts/stops serving them accordingly.

If there are no permissions to list/watch the namespaces, the operator falls back to the list of provided namespaces "as is", assuming they exist. Namespace patterns do not work in this case; only the specific namespaces do (which means, all namespaces with the `,*?!` characters are excluded).

If a namespace does not exist, Kubernetes permits watching over it anyway. The only difference is when the resource watching starts: if the permissions are sufficient, the watching starts only after the namespace is created; if not sufficient,

the watching starts immediately (for an unexistent namespace) and the resources will be served once that namespace is created.

22.2 Cluster-wide

To serve the resources in the whole cluster:

```
kopf run --all-namespaces ...
kopf run -A ...
```

In that case, the operator does not monitor the namespaces in the cluster, and uses different K8s API URLs to list/watch the objects cluster-wide.

CHAPTER

TWENTYTHREE

IN-MEMORY CONTAINERS

Kopf provides several ways of storing and exchanging the data in-memory between handlers and operators.

23.1 Resource memos

Every resource handler gets a *memo* kwarg of type *kopf.Memo*. It is an in-memory container for arbitrary runtime-only keys-values. The values can be accessed as either object attributes or dictionary keys.

The memo is shared by all handlers of the same individual resource (not of the resource kind, but a resource object). If the resource is deleted and re-created with the same name, the memo is also re-created (technically, it is a new resource).

```
import kopf

@kopf.on.event('KopfExample')
def pinged(memo: kopf.Memo, **_):
    memo.counter = memo.get('counter', 0) + 1

@kopf.timer('KopfExample', interval=10)
def tick(memo: kopf.Memo, logger, **_):
    logger.info(f"{memo.counter} events have been received in 10 seconds.")
    memo.counter = 0
```

23.2 Operator memos

In the operator handlers, such as the operator startup/cleanup, liveness probes, credentials retrieval, and everything else not specific to resources, *memo* points to the operator's global container for arbitrary values.

The per-operator container can be either populated in the startup handlers, or passed from outside of the operator when *Embedding* is used, or both:

```
import kopf
import queue
import threading

@kopf.on.startup()
def start_background_worker(memo: kopf.Memo, **_):
    memo.my_queue = queue.Queue()
    memo.my_thread = threading.Thread(target=background, args=(memo.my_queue,))
    memo.my_thread.start()
```

(continues on next page)

(continued from previous page)

```
@kopf.on.cleanup()
def stop_background_worker(memo: kopf.Memo, **_):
    memo['my_queue'].put(None)
    memo['my_thread'].join()

def background(queue: queue.Queue):
    while True:
        item = queue.get()
        if item is None:
            break
        else:
            print(item)
```

Note: For code quality and style consistency, it is recommended to use the same approach when accessing the stored values. The mixed style here is for demonstration purposes only.

The operator's memo is later used to populate the per-resource memos. All keys & values are shallow-copied into each resource's memo, where they can be mixed with the per-resource values:

```
# ... continued from the previous example.
@kopf.on.event('KopfExample')
def pinged(memo: kopf.Memo, namespace: str, name: str, **_):
    if not memo.get('is_seen'):
        memo.my_queue.put(f"{namespace}/{name}")
        memo.is_seen = True
```

Any changes to the operator's container since the first appearance of the resource are **not** replicated to the existing resources' containers, and are not guaranteed to be seen by the new resources (even if they are now).

However, due to shallow copying, the mutable objects (lists, dicts, and even custom instances of *kopf.Memo* itself) in the operator's container can be modified from outside, and these changes will be seen in all individual resource handlers & daemons which use their per-resource containers.

23.3 Custom memo classes

For embedded operators (*Embedding*), it is possible to use any class for memos. It is not even required to inherit from *kopf.Memo*.

There are 2 strict requirements:

- The class must be supported by all involved handlers that use it.
- The class must support shallow copying via `copy.copy()` (`__copy__()`).

The latter is used to create per-resource memos from the operator's memo. To have one global memo for all individual resources, redefine the class to return `self` when requested to make a copy, as shown below:

```
import asyncio
import dataclasses
import kopf
```

(continues on next page)

(continued from previous page)

```
@dataclasses.dataclass()
class CustomContext:
    create_tpl: str
    delete_tpl: str

    def __copy__(self) -> "CustomContext":
        return self

@kopf.on.create('kopfexamples')
def create_fn(memo: CustomContext, **kwargs):
    print(memo.create_tpl.format(**kwargs))

@kopf.on.delete('kopfexamples')
def delete_fn(memo: CustomContext, **kwargs):
    print(memo.delete_tpl.format(**kwargs))

if __name__ == '__main__':
    kopf.configure(verbose=True)
    asyncio.run(kopf.operator(
        memo=CustomContext(
            create_tpl="Hello, {name}!",
            delete_tpl="Good bye, {name}!",
        ),
    ))
```

In all other regards, the framework does not use memos for its own needs and passes them through the call stack to the handlers and daemons "as is".

This advanced feature is not available for operators executed via `kopf run`.

23.4 Limitations

All in-memory values are lost on operator restarts; there is no persistence.

The in-memory containers are recommended only for ephemeral objects scoped to the process lifetime, such as concurrency primitives: locks, tasks, threads… For persistent values, use the status stanza or annotations of the resources.

Essentially, the operator's memo is not much different from global variables (unless 2+ embedded operator tasks are running there) or asyncio contextvars, except that it provides the same interface as for the per-resource memos.

See also:

In-memory indexing — other in-memory structures with similar limitations.

CHAPTER

TWENTYFOUR

IN-MEMORY INDEXING

Indexers automatically maintain in-memory overviews of resources (indices), grouped by keys that are usually calculated based on these resources.

The indices can be used for cross-resource awareness: e.g., when a resource of kind X is changed, it can get all the information about all resources of kind Y without talking to the Kubernetes API. Under the hood, the centralised watch-streams —one per resource kind— are more efficient in gathering the information than individual listing requests.

24.1 Index declaration

Indices are declared with a `@kopf.index` decorator on an indexing function (all standard filters are supported — see *Filtering*):

```
import kopf

@kopf.index('pods')
def my_idx(**_):
    ...
```

The name of the function or its `id=` option is the index's name.

The indices are then available to all resource- and operator-level handlers as the direct kwargs named the same as the index (type hints are optional):

```
import kopf

# ... continued from previous examples:
@kopf.timer('KopfExample', interval=5)
def tick(my_idx: kopf.Index, **_):
    ...

@kopf.on.probe()
def metric(my_idx: kopf.Index, **_):
    ...
```

When a resource is created or starts matching the filters, it is processed by all relevant indexing functions, and the result is put into the indices.

When a previously indexed resource is deleted or stops matching the filters, all associated values are removed (so are all empty collections after this — to keep the indices clean).

See also:

Health-checks for probing handlers in the example above.

24.2 Index structure

An index is always a read-only *mapping* of type *kopf.Index* with arbitrary keys leading to *collections* of type *kopf.Store*, which in turn contain arbitrary values generated by the indexing functions. The index is initially empty. The collections are never empty (empty collections are removed when the last item in them is removed).

For example, if several individual resources return the following results from the same indexing function, then the index gets the following structure (shown in the comment below the code):

```
return {'key1': 'valueA'}  # 1st
return {'key1': 'valueB'}  # 2nd
return {'key2': 'valueC'}  # 3rd
# {'key1': ['valueA', 'valueB'],
#  'key2': ['valueC']}
```

The indices are not nested. The 2nd-level mapping in the result is stored as a regular value:

```
return {'key1': 'valueA'}  # 1st
return {'key1': 'valueB'}  # 2nd
return {'key2': {'key3': 'valueC'}}  # 3rd
# {'key1': ['valueA', 'valueB'],
#  'key2': [{'key3': 'valueC'}]}
```

24.3 Index content

When an indexing function returns a `dict` (strictly `dict`! not a generic mapping, not even a descendant of `dict`, such as *kopf.Memo*), it is merged into the index under the key taken from the result:

```
import kopf

@kopf.index('pods')
def string_keys(namespace, name, **_):
    return {namespace: name}
    # {'namespace1': ['pod1a', 'pod1b', ...],
    #  'namespace2': ['pod2a', 'pod2b', ...],
    #   ...]
```

Multi-value keys are possible with e.g. tuples or other hashable types:

```
import kopf

@kopf.index('pods')
def tuple_keys(namespace, name, **_):
    return {(namespace, name): 'hello'}
    # {('namespace1', 'pod1a'): ['hello'],
    #  ('namespace1', 'pod1b'): ['hello'],
    #  ('namespace2': 'pod2a'): ['hello'],
```

(continues on next page)

(continued from previous page)

```
    #  ('namespace2', 'pod2b'): ['hello'],
    #   ...}
```

Multiple keys can be returned at once for a single resource. They are all merged into their relevant places in the index:

```
import kopf

@kopf.index('pods')
def by_label(labels, name, **_):
    return {(label, value): name for label, value in labels.items()}
    # {('label1', 'value1a'): ['pod1', 'pod2', ...],
    #  ('label1', 'value1b'): ['pod3', 'pod4', ...],
    #  ('label2', 'value2a'): ['pod5', 'pod6', ...],
    #  ('label2', 'value2b'): ['pod1', 'pod3', ...],
    #   ...}

@kopf.timer('kex', interval=5)
def tick(by_label: kopf.Index, **_):
    print(list(by_label.get(('label2', 'value2b'), [])))
    # ['pod1', 'pod3']
    for podname in by_label.get(('label2', 'value2b'), []):
        print(f"==> {podname}")
    # ==> pod1
    # ==> pod3
```

Note the multiple occurrences of some pods because they have two or more labels. But they never repeat within the same label — labels can have only one value.

24.4 Recipes

24.4.1 Unindexed collections

When an indexing function returns a non-`dict` — i.e. strings, numbers, tuples, lists, sets, memos, arbitrary objects except `dict` — then the key is assumed to be `None` and a flat index with only one key is constructed. The resources are not indexed, but rather collected under the same key (which is still considered as indexing):

```
import kopf

@kopf.index('pods')
def pod_names(name: str, **_):
    return name
    # {None: ['pod1', 'pod2', ...]}
```

Other types and complex objects returned from the indexing function are stored "as is" (i.e. with no special treatment):

```
import kopf

@kopf.index('pods')
def container_names(spec: kopf.Spec, **_):
    return {container['name'] for container in spec.get('containers', [])}
    # {None: [{'main1', 'sidecar2'}, {'main2'}, ...]}
```

24.4.2 Enumerating resources

If the goal is not to store any payload but to only list the existing resources, then index the resources' identities (usually, their namespaces and names).

One way is to only collect their identities in a flat collection – in case you need mostly to iterate over all of them without key lookups:

```
import kopf

@kopf.index('pods')
def pods_list(namespace, name, **_):
    return namespace, name
    # {None: [('namespace1', 'pod1a'),
    #         ('namespace1', 'pod1b'),
    #         ('namespace2', 'pod2a'),
    #         ('namespace2', 'pod2b'),
    #           ...]}

@kopf.timer('kopfexamples', interval=5)
def tick_list(pods_list: kopf.Index, **_):
    for ns, name in pods_list.get(None, []):
        print(f"{ns}::{name}")
```

Another way is to index them by keys — when index lookups are going to happen more often than index iterations:

```
import kopf

@kopf.index('pods')
def pods_dict(namespace, name, **_):
    return {(namespace, name): None}
    # {('namespace1', 'pod1a'): [None],
    #  ('namespace1', 'pod1b'): [None],
    #  ('namespace2', 'pod2a'): [None],
    #  ('namespace2', 'pod2b'): [None],
    #   ...}

@kopf.timer('kopfexamples', interval=5)
def tick_dict(pods_dict: kopf.Index, spec: kopf.Spec, namespace: str, **_):
    monitored_namespace = spec.get('monitoredNamespace', namespace)
    for ns, name in pods_dict:
        if ns == monitored_namespace:
            print(f"in {ns}: {name}")
```

24.4.3 Mirroring resources

To store the whole resource or its essential parts, return them explicitly:

```
import kopf

@kopf.index('deployments')
def whole_deployments(name: str, namespace: str, body: kopf.Body, **_):
    return {(namespace, name): body}

@kopf.timer('kopfexamples', interval=5)
def tick(whole_deployments: kopf.Index, **_):
    deployment, *_ = whole_deployments[('kube-system', 'coredns')]
    actual = deployment.status.get('replicas')
    desired = deployment.spec.get('replicas')
    print(f"{deployment.meta.name}: {actual}/{desired}")
```

Note: Mind the memory consumption on large clusters and/or overly verbose objects. Especially mind the memory consumption for "managed fields" (see kubernetes/kubernetes#90066).

24.4.4 Indices of indices

Iterating over all keys of the index can be slow (especially if there are many keys: e.g. with thousands of pods). For that case, an index of an index can be built: with one primary indexing containing the real values to be used, while the other secondary index only contains the keys of the primary index (full or partial).

By looking up a single key in the secondary index, the operator can directly get or indirectly reconstruct all the necessary keys in the primary index instead of iterating over the primary index with filtering.

For example, we want to get all container names of all pods in a namespace. In that case, the primary index will index containers by pods' namespaces+names, while the secondary index will index pods' names by namespaces only:

```
import kopf

@kopf.index('pods')
def primary(namespace, name, spec, **_):
    container_names = {container['name'] for container in spec['containers']}
    return {(namespace, name): container_names}
    # {('namespace1', 'pod1a'): [{'main'}],
    #  ('namespace1', 'pod1b'): [{'main', 'sidecar'}],
    #  ('namespace2', 'pod2a'): [{'main'}],
    #  ('namespace2', 'pod2b'): [{'the-only-one'}],
    #   ...}

@kopf.index('pods')
def secondary(namespace, name, **_):
    return {namespace: name}
    # {'namespace1': ['pod1a', 'pod1b'],
    #  'namespace2': ['pod2a', 'pod2b'],
    #   ...}

@kopf.timer('kopfexamples', interval=5)
```

(continues on next page)

(continued from previous page)

```
def tick(primary: kopf.Index, secondary: kopf.Index, spec: kopf.Spec, **_):
    namespace_containers = set()
    monitored_namespace = spec.get('monitoredNamespace', 'default')
    for pod_name in secondary.get(monitored_namespace, []):
        reconstructed_key = (monitored_namespace, pod_name)
        pod_containers, *_ = primary[reconstructed_key]
        namespace_containers |= pod_containers
    print(f"containers in {monitored_namespace}: {namespace_containers}")
    # containers in namespace1: {'main', 'sidecar'}
    # containers in namespace2: {'main', 'the-only-one'}
```

However, such complicated structures and such performance requirements are rare. For simplicity and performance, nested indices are not directly provided by the framework as a feature, only as this tip based on other official features.

24.5 Conditional indexing

Besides the usual filters (see *Filtering*), the resources can be skipped from indexing by returning `None` (Python's default for no-result functions).

If the indexing function returns `None` or does not return anything, its result is ignored and not indexed. The existing values in the index are preserved as they are (this is also the case when unexpected errors happen in the indexing function with the errors mode set to `IGNORED`):

```
import kopf

@kopf.index('pods')
def empty_index(**_):
    pass
    # {}
```

However, if the indexing function returns a dict with `None` as values, such values are indexed as usually (they are not ignored). `None` values can be used as placeholders when only the keys are sufficient; otherwise, indices and collections with no values left in them are removed from the index:

```
import kopf

@kopf.index('pods')
def index_of_nones(**_):
    return {'key': None}
    # {'key': [None, None, ...]}
```

24.6 Errors in indexing

The indexing functions are supposed to be fast and non-blocking, as they are capable of delaying the operator startup and resource processing. For this reason, in case of errors in handlers, the handlers are never retried.

Arbitrary exceptions with `errors=IGNORED` (the default) make the framework ignore the error and keep the existing indexed values (which are now stale). It means that the new values are expected to appear soon, but the old values are good enough meanwhile (which is usually highly probable). This is the same as returning `None`, except that the exception's stack trace is logged too:

```
import kopf

@kopf.index('pods', errors=kopf.ErrorsMode.IGNORED)  # the default
def fn1(**_):
    raise Exception("Keep the stale values, if any.")
```

kopf.PermanentError and arbitrary exceptions with `errors=PERMANENT` remove any existing indexed values and the resource's keys from the index, and exclude the failed resource from indexing by this index in the future (so that even the indexing function is not invoked for them):

```
import kopf

@kopf.index('pods', errors=kopf.ErrorsMode.PERMANENT)
def fn1(**_):
    raise Exception("Excluded forever.")

@kopf.index('pods')
def fn2(**_):
    raise kopf.PermamentError("Excluded forever.")
```

kopf.TemporaryError and arbitrary exceptions with `errors=TEMPORARY` remove any existing indexed values and the resource's keys from the index, and exclude the failed resource from indexing for the specified duration (via the error's `delay` option; set to `0` or `None` for no delay). It is expected that the resource could be reindexed in the future, but right now, problems are preventing this from happening:

```
import kopf

@kopf.index('pods', errors=kopf.ErrorsMode.TEMPORARY)
def fn1(**_):
    raise Exception("Excluded for 60s.")

@kopf.index('pods')
def fn2(**_):
    raise kopf.TemporaryError("Excluded for 30s.", delay=30)
```

In the "temporary" mode, the decorator's options for error handling are used: the `backoff=` is a default delay before the resource can be re-indexed (the default is 60 seconds; for no delay, use `0` explicitly); the `retries=` and `timeout=` are the limit of retries and the overall duration since the first failure until the resource will be marked as permanently excluded from indexing (unless it succeeds at some point).

The handler's kwargs *retry*, *started*, *runtime* report the retrying attempts since the first indexing failure. Successful indexing resets all the counters/timeouts and the retrying state is not stored (to save memory).

The same as with regular handlers (*Error handling*), Kopf's error classes (expected errors) only log a short message, while arbitrary exceptions (unexpected errors) also dump their stack traces.

This matches the semantics of regular handlers but with in-memory specifics.

Warning: There is no good out-of-the-box default mode for error handling: any kind of errors in the indexing functions means that the index becomes inconsistent with the actual state of the cluster and its resources: the entries for matching resources are either "lost" (permanent or temporary errors), or contain possibly outdated/stale values (ignored errors) — all of these cases are misinformation about the actual state of the cluster.

The default mode is chosen to reduce the index changes and reindexing in case of frequent errors — by not making any changes to the index. Besides, the stale values can still be relevant and useful to some extent.

For two other cases, the operator developers have to explicitly accept the risks by setting `errors=` if the operator can afford to lose the keys.

24.7 Kwargs safety

Indices that are injected into kwargs, overwrite any kwargs of the framework, existing and those to be added later. This guarantees that the new framework versions will not break an operator if new kwargs are added with the same name as the existing indices.

In this case, the trade-off is that the handlers cannot use the new features until their indices are renamed to something else. Since the new features are new, the old operator's code does not use them, so it is backwards compatible.

To reduce the probability of name collisions, keep these conventions in mind when naming indices (they are fully optional and for convenience only):

- System kwargs are usually one-word; name your indices with 2+ words.
- System kwargs are usually singular (not always); name the indices as plurals.
- System kwargs are usually nouns; using abbreviations or prefixes/suffixes (e.g. `cnames`, `rpods`) would reduce the probability of collisions.

24.8 Performance

Indexing can be a CPU- & RAM-consuming operation. The data structures behind indices are chosen to be as efficient as possible:

- The index's lookups are O(1) — as in Python's `dict`.
- The store's updates/deletions are O(1) – a `dict` is used internally.
- The overall updates/deletions are O(k), where "k" is the number of keys per object (not of all keys!), which is fixed in most cases, so it is O(1).

Neither the number of values stored in the index nor the overall amount of keys affect its performance (in theory).

Some performance can be lost on additional method calls of the user-facing mappings/collections made to hide the internal `dict` structures. It is assumed to be negligible compared to the overall code overhead.

24.9 Guarantees

If an index is declared, there is no need to additionally pre-check for its existence — the index exists immediately even if it contains no resources.

The indices are guaranteed to be fully pre-populated before any other resource-related handlers are invoked in the operator. As such, even the on-creation handlers or raw event handlers are guaranteed to have the complete indexed overview of the cluster, not just partially populated to the moment when they happened to be triggered.

There is no such guarantee for the operator handlers, such as startup/cleanup, authentication, health probing, and for the indexing functions themselves: the indices are available in kwargs but can be empty or partially populated in the operator's startup and index pre-population stage. This can affect the cleanup/login/probe handlers if they are invoked at that stage.

Though, the indices are safe to be passed to threads/tasks for later processing if such threads/tasks are started from the before-mentioned startup handlers.

24.10 Limitations

All in-memory values are lost on operator restarts; there is no persistence. In particular, the indices are fully recalculated on operator restarts during the initial listing of the resources (equivalent to `@kopf.on.event`).

On large clusters with thousands of resources, the initial index population can take time, so the operator's processing will be delayed regardless of whether the handlers do use the indices or they do not (the framework cannot know this for sure).

See also:

In-memory containers — other in-memory structures with similar limitations.

See also:

Indexers and indices are conceptually similar to client-go's indexers – with all the underlying components implemented inside of the framework ("batteries included").

CHAPTER

TWENTYFIVE

ADMISSION CONTROL

Admission hooks are callbacks from Kubernetes to the operator before the resources are created or modified. There are two types of hooks:

- Validating admission webhooks.
- Mutating admission webhooks.

For more information on the admission webhooks, see the Kubernetes documentation: Dynamic Admission Control.

25.1 Dependencies

To minimize Kopf's footprint in production systems, it does not include heavy-weight dependencies needed only for development, such as SSL cryptography and certificate generation libraries. For example, Kopf's footprint with critical dependencies is 8.8 MB, while `cryptography` would add 8.7 MB; `certbuilder` adds "only" 2.9 MB.

To use all features of development-mode admission webhook servers and tunnels, you have to install Kopf with an extra:

```
pip install kopf[dev]
```

If this extra is not installed, Kopf will not generate self-signed certificates and will run either with HTTP only or with externally provided certificates.

Also, without this extra, Kopf will not be able to establish Ngrok tunnels. Though, it will be able to use K3d & Minikube servers with magic hostnames.

Any attempt to run it in a mode with self-signed certificates or tunnels will raise a startup-time error with an explanation and suggested actions.

25.2 Validation handlers

```
import kopf

@kopf.on.validate('kopfexamples')
def say_hello(warnings: list[str], **_):
    warnings.append("Verified with the operator's hook.")

@kopf.on.validate('kopfexamples')
def check_numbers(spec, **_):
    if not isinstance(spec.get('numbers', []), list):
```

(continues on next page)

(continued from previous page)

```
        raise kopf.AdmissionError("Numbers must be a list if present.")

@kopf.on.validate('kopfexamples')
def convertible_numbers(spec, warnings, **_):
    if isinstance(spec.get('numbers', []), list):
        for val in spec.get('numbers', []):
            if not isinstance(val, float):
                try:
                    float(val)
                except ValueError:
                    raise kopf.AdmissionError(f"Cannot convert {val!r} to a number.")
                else:
                    warnings.append(f"{val!r} is not a number but can be converted.")

@kopf.on.validate('kopfexamples')
def numbers_range(spec, **_):
    if isinstance(spec.get('numbers', []), list):
        if not all(0 <= float(val) <= 100 for val in spec.get('numbers', [])):
            raise kopf.AdmissionError("Numbers must be below 0..100.", code=499)
```

Each handler is mapped to its dedicated admission webhook and an endpoint so that all handlers are executed in parallel independently of each other. They must not expect that other checks are already performed by other handlers; if such logic is needed, make it as one handler with a sequential execution.

25.3 Mutation handlers

To mutate the object, modify the *patch*. Changes to *body*, *spec*, etc, will not be remembered (and are not possible):

```
import kopf

@kopf.on.mutate('kopfexamples')
def ensure_default_numbers(spec, patch, **_):
    if 'numbers' not in spec:
        patch.spec['numbers'] = [1, 2, 3]

@kopf.on.mutate('kopfexamples')
def convert_numbers_if_possible(spec, patch, **_):
    if 'numbers' in spec and isinstance(spec.get('numbers'), list):
        patch.spec['numbers'] = [_maybe_number(v) for v in spec['numbers']]

def _maybe_number(v):
    try:
        return float(v)
    except ValueError:
        return v
```

The semantics is the same or as close as possible to the Kubernetes API's one. `None` values will remove the relevant keys.

Under the hood, the patch object will remember each change and will return a JSONPatch structure to Kubernetes.

25.4 Handler options

Handlers have a limited capability to inform Kubernetes about its behaviour. The following options are supported:

`persistent` (`bool`) webhooks will not be removed from the managed configurations on exit; non-persisted webhooks will be removed if possible. Such webhooks will prevent all admissions even when the operator is down. This option has no effect if there is no managed configuration. The webhook cleanup only happens on graceful exits; on forced exits, even non-persisted webhooks might be persisted and block the admissions.

`operation` (`str`) will configure this handler/webhook to be called only for a specific operation. For multiple operations, add several decorators. Possible values are `"CREATE"`, `"UPDATE"`, `"DELETE"`, `"CONNECT"`. The default is `None`, i.e. all operations (equivalent to `"*"`).

`subresource` (`str`) will only react when to the specified subresource. Usually it is `"status"` or `"scale"`, but can be anything else. The value `None` means that only the main resource body will be checked. The value `"*"` means that both the main body and any subresource are checked. The default is `None`, i.e. only the main body to be checked.

`side_effects` (`bool`) tells Kubernetes that the handler can have side effects in non-dry-run mode. In dry-run mode, it must have no side effects. The dry-run mode is passed to the handler as a *dryrun* kwarg. The default is `False`, i.e. the handler has no side effects.

`ignore_failures` (`bool`) marks the webhook as tolerant to errors. This includes errors of the handler itself (disproved admissions), so as HTTP/TCP communication errors when apiservers talk to the webhook server. By default, an inaccessible or rejecting webhook blocks the admission.

The developers can use regular *Filtering*. In particular, the `labels` will be passed to the webhook configuration as `.webhooks.*.objectSelector` for optimization purposes: so that admissions are not even sent to the webhook server if it is known that they will be filtered out and therefore allowed.

Server-side filtering supports everything except callbacks: i.e., `"strings"`, `kopf.PRESENT` and `kopf.ABSENT` markers. The callbacks will be evaluated after the admission review request is received.

> **Warning:** Be careful with the builtin resources and admission hooks. If a handler is broken or misconfigured, it can prevent creating those resources, e.g. pods, in the whole cluster. This will render the cluster unusable until the configuration is manually removed.
>
> Start the development in local clusters, validating/mutating the custom resources first, and enable `ignore_errors` initially. Enable the strict mode of the handlers only when stabilised.

25.5 In-memory containers

Kopf provides *In-memory containers* for each resource. However, webhooks can happen before a resource is created. This affects how the memos work.

For update and deletion requests, the actual memos of the resources are used.

For the admission requests on resource creation, a memo is created and discarded immediately. It means that the creation's memos are useless at the moment.

This can change in the future: the memos of resource creation attempts will be preserved for a limited but short time (configurable), so that the values could be shared between the admission and the handling, but so that there are no memory leaks if the resource never succeeds in admission.

25.6 Admission warnings

Starting with Kubernetes 1.19 (check with `kubectl version`), admission warnings can be returned from admission handlers.

To populate warnings, accept a **mutable** *warnings* (`list[str]`) and add strings to it:

```
import kopf

@kopf.on.validate('kopfexamples')
def ensure_default_numbers(spec, warnings: list[str], **_):
    if spec.get('field') == 'value':
        warnings.append("The default value is used. It is okay but worth changing.")
```

The admission warnings look like this (requires kubectl 1.19+):

```
$ kubectl create -f examples/obj.yaml
Warning: The default value is used. It is okay but worth changing.
kopfexample.kopf.dev/kopf-example-1 created
```

Note: Despite Kopf's intention to utilise Python's native features that semantically map to Kubernetes's or operators' features, Python StdLib's `warnings` is not used for admission warnings (the initial idea was to catch `UserWarning` and `warnings.warn("...")` calls and return them as admission warnings).

The StdLib's module is documented as thread-unsafe (therefore, task-unsafe) and requires hacking the global state which might affect other threads and/or tasks – there is no clear way to do this consistently.

This may be revised in the future and provided as an additional feature.

25.7 Admission errors

Unlike with regular handlers and their error handling logic (*Error handling*), the webhooks cannot do retries or backoffs. So, the `backoff=`, `errors=`, `retries=`, `timeout=` options are not accepted on the admission handlers.

A special exception *kopf.AdmissionError* is provided to customize the status code and the message of the admission review response.

All other exceptions, including *kopf.PermanentError* and *kopf.TemporaryError*, equally fail the admission (be that validating or mutating admission). However, they return the general HTTP code 500 (non-customisable).

One and only one error is returned to the user who make an API request. In cases when Kubernetes makes several parallel requests to several webhooks (typically with managed webhook configurations, the fastest error is used). Within Kopf (usually with custom webhook servers/tunnels or self-made non-managed webhook configurations), errors are prioritised: first, admission errors, then permanent errors, then temporary errors, then arbitrary errors are used to select the only error to report in the admission review response.

```
@kopf.on.validate('kopfexamples')
def validate1(spec, **_):
    if spec.get('field') == 'value':
        raise kopf.AdmissionError("Meh! I don't like it. Change the field.", code=400)
```

The admission errors look like this (manually indented for readability):

```
$ kubectl create -f examples/obj.yaml
Error from server: error when creating "examples/obj.yaml":
    admission webhook "validate1.auto.kopf.dev" denied the request:
        Meh! I don't like it. Change the field.
```

Note that Kubernetes executes multiple webhooks in parallel. The first one to return the result is the one and the only shown; other webhooks are not shown even if they fail with useful messages. With multiple failing admissions, the message will be varying on each attempt.

25.8 Webhook management

Admission (both for validation and for mutation) only works when the cluster has special resources created: either `kind: ValidatingWebhookConfiguration` or `kind: MutatingWebhookConfiguration` or both. Kopf can automatically manage the webhook configuration resources in the cluster if it is given RBAC permissions to do so.

To manage the validating/mutating webhook configurations, Kopf requires the following RBAC permissions in its service account (see *Deployment*):

```yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
rules:
  - apiGroups: [admissionregistration.k8s.io/v1, admissionregistration.k8s.io/v1beta1]
    resources: [validatingwebhookconfigurations, mutatingwebhookconfigurations]
    verbs: [create, patch]
```

By default, configuration management is disabled (for safety and stability). To enable, set the name of the managed configuration objects:

```python
@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.admission.managed = 'auto.kopf.dev'
```

Multiple records for webhooks will be added or removed for multiple resources to those configuration objects as needed. Existing records will be overwritten. If the configuration resource is absent, it will be created (but at most one for validating and one for mutating configurations).

Kopf manages the webhook configurations according to how Kopf itself believes it is sufficient to achieve the goal. Many available Kubernetes features are not covered by this management. To use these features and control the configuration with precision, operator developers can disable the automated management and take care of the configuration manually.

25.9 Servers and tunnels

Kubernetes admission webhooks are designed to be passive rather than active (from the operator's point of view; vice versa from Kubernetes's point of view). It means, the webhooks must passively wait for requests via an HTTPS endpoint. There is currently no official way how an operator can actively pull or poll the admission requests and send the responses back (as it is done for all other resource changes streamed via the Kubernetes API).

It is typically non-trivial to forward the requests from a remote or isolated cluster to a local host machine where the operator is running for development.

However, one of Kopf's main promises is to work the same way both in-cluster and on the developers' machines. It cannot be made "the same way" for webhooks, but Kopf attempts to make these modes similar to each other code-wise.

To fulfil its promise, Kopf delegates this task to webhook servers and tunnels, which are capable of receiving the webhook requests, marshalling them to the handler callbacks, and then returning the results to Kubernetes.

Due to numerous ways of how the development and production environments can be configured, Kopf does not provide a default configuration for a webhook server, so it must be set by the developer:

```
@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    if os.environ.get('ENVIRONMENT') is None:
        # Only as an example:
        settings.admission.server = kopf.WebhookK3dServer(port=54321)
        settings.admission.managed = 'auto.kopf.dev'
    else:
        # Assuming that the configuration is done manually:
        settings.admission.server = kopf.WebhookServer(addr='0.0.0.0', port=8080)
        settings.admission.managed = 'auto.kopf.dev'
```

If there are admission handlers present and no webhook server/tunnel configured, the operator will fail at startup with an explanatory message.

Kopf provides several webhook servers and tunnels out of the box, each with its configuration parameters (see their descriptions):

Webhook servers listen on an HTTPS port locally and handle requests.

- *kopf.WebhookServer* is helpful for local development and `curl` and a Kubernetes cluster that runs directly on the host machine and can access it. It is also used internally by most tunnels for a local target endpoint.
- *kopf.WebhookK3dServer* is for local K3d/K3s clusters (even in a VM), accessing the server via a magical hostname `host.k3d.internal`.
- *kopf.WebhookMinikubeServer* for local Minikube clusters (even in VMs), accessing the server via a magical hostname `host.minikube.internal`.

Webhook tunnels forward the webhook requests through external endpoints usually to a locally running *webhook server*.

- *kopf.WebhookNgrokTunnel* established a tunnel through ngrok.

For ease of use, the cluster type can be recognised automatically in some cases:

- *kopf.WebhookAutoServer* runs locally, detects Minikube & K3s, and uses them via their special hostnames. If it cannot detect the cluster type, it runs a simple local webhook server. The auto-server never tunnels.
- *kopf.WebhookAutoTunnel* attempts to use an auto-server if possible. If not, it uses one of the available tunnels (currently, only ngrok). This is the most universal way to make any environment work.

Note: External tunnelling services usually limit the number of requests. For example, ngrok has a limit of 40 requests per minute on a free plan.

The services also usually provide paid subscriptions to overcome that limit. It might be a wise idea to support the service you rely on with some money. If that is not an option, you can implement free tunnelling your way.

Note: A reminder: using development-mode tunnels and self-signed certificates requires extra dependencies: `pip install kopf[dev]`.

25.10 Authenticate apiservers

There are many ways how webhook clients (Kubernetes's apiservers) can authenticate against webhook servers (the operator's webhooks), and even more ways to validate the supplied credentials.

More on that, apiservers cannot be configured to authenticate against webhooks dynamically at runtime, as this requires control-plane configs, which are out of reach of Kopf.

For simplicity, Kopf does not authenticate webhook clients.

However, Kopf's built-in webhook servers & tunnels extract the very basic request information and pass it to the admission handlers for additional verification and possibly for authentification:

- *headers* (`Mapping[str, str]`) contains all HTTPS headers, including `Authorization: Basic ...`, `Authorization: Bearer ...`.
- *sslpeer* (`Mapping[str, Any]`) contains the SSL peer information as returned by `ssl.SSLSocket.getpeercert()` or `None` if no proper SSL certificate is provided by a client (i.e. by apiservers talking to webhooks).

An example of headers:

```
{'Host': 'localhost:54321',
 'Authorization': 'Basic dXNzc2VyOnBhc3Nzdw==',  # base64("ussser:passsw")
 'Content-Length': '844',
 'Content-Type': 'application/x-www-form-urlencoded'}
```

An example of a self-signed peer certificate presented to `sslpeer`:

```
{'subject': ((('commonName', 'Example Common Name'),),
             (('emailAddress', 'example@kopf.dev'),)),
 'issuer': ((('commonName', 'Example Common Name'),),
            (('emailAddress', 'example@kopf.dev'),)),
 'version': 1,
 'serialNumber': 'F01984716829537E',
 'notBefore': 'Mar  7 17:12:20 2021 GMT',
 'notAfter': 'Mar  7 17:12:20 2022 GMT'}
```

To reproduce these examples without configuring the Kubernetes apiservers but only Kopf & CLI tools, do the following:

Step 1: Generate a self-signed ceritificate to be used as a client certificate:

```
openssl req -x509 -newkey rsa:2048 -keyout client-key.pem -out client-cert.pem -days 365␣
↪-nodes
# Country Name (2 letter code) []:
# State or Province Name (full name) []:
# Locality Name (eg, city) []:
# Organization Name (eg, company) []:
# Organizational Unit Name (eg, section) []:
# Common Name (eg, fully qualified host name) []:Example Common Name
# Email Address []:example@kopf.dev
```

Step 2: Start an operator with the certificate as a CA (for simplicity; in normal setups, there is a separate CA, which signs the client certificates; explaining this topic is beyond the scope of this framework's documentation):

```
import kopf

@kopf.on.startup()
def config(settings: kopf.OperatorSettings, **_):
    settings.admission.managed = 'auto.kopf.dev'
    settings.admission.server = kopf.WebhookServer(cafile='client-cert.pem')

@kopf.on.validate('kex')
def show_auth(headers, sslpeer, **_):
    print(f'{headers=}')
    print(f'{sslpeer=}')
```

Step 3: Save the admission review payload into a local file:

```
cat >review.json << EOF
{
  "kind": "AdmissionReview",
  "apiVersion": "admission.k8s.io/v1",
  "request": {
    "uid": "1ca13837-ad60-4c9e-abb8-86f29d6c0e84",
    "kind": {"group": "kopf.dev", "version": "v1", "kind": "KopfExample"},
    "resource": {"group": "kopf.dev", "version": "v1", "resource": "kopfexamples"},
    "requestKind": {"group": "kopf.dev", "version": "v1", "kind": "KopfExample"},
    "requestResource": {"group": "kopf.dev", "version": "v1", "resource": "kopfexamples"}
↪,
    "name": "kopf-example-1",
    "namespace": "default",
    "operation": "CREATE",
    "userInfo": {"username": "admin", "uid": "admin", "groups": ["system:masters",
↪"system:authenticated"]},
    "object": {
      "apiVersion": "kopf.dev/v1",
      "kind": "KopfExample",
      "metadata": {"name": "kopf-example-1", "namespace": "default"}
    },
    "oldObject": null,
    "dryRun": true
  }
}
EOF
```

Step 4: Send the admission review payload to the operator's webhook server using the generated client certificate, observe the client identity printed to stdout by the webhook server and returned in the warnings:

```
curl --insecure --cert client-cert.pem --key client-key.pem https://
↪ussser:passsw@localhost:54321 -d @review.json
# {"apiVersion": "admission.k8s.io/v1", "kind": "AdmissionReview",
#  "response": {"uid": "1ca13837-ad60-4c9e-abb8-86f29d6c0e84",
#               "allowed": true,
#               "warnings": ["SSL peer is Example Common Name."]}}
```

Using this data, operator developers can implement servers/tunnels with custom authentication methods when and if needed.

25.11 Debugging with SSL

Kubernetes requires that the webhook URLs are always HTTPS, never HTTP. For this reason, Kopf runs the webhook servers/tunnels with HTTPS by default.

If a webhook server is configured without a server certificate, a self-signed certificate is generated at startup, and only HTTPS is served.

```
@kopf.on.startup()
def config(settings: kopf.OperatorSettings, **_):
    settings.admission.server = kopf.WebhookServer()
```

That endpoint can be accessed directly with `curl`:

```
curl --insecure https://localhost:54321 -d @review.json
```

It is possible to store the generated certificate itself and use as a CA:

```
@kopf.on.startup()
def config(settings: kopf.OperatorSettings, **_):
    settings.admission.server = kopf.WebhookServer(cadump='selfsigned.pem')
```

```
curl --cacert selfsigned.pem https://localhost:54321 -d @review.json
```

For production, a properly generated certificate should be used. The CA, if not specified, is assumed to be in the default trust chain. This applies to all servers: *kopf.WebhookServer*, *kopf.WebhookK3dServer*, etc.

```
@kopf.on.startup()
def config(settings: kopf.OperatorSettings, **_):
    settings.admission.server = kopf.WebhookServer(
        cafile='ca.pem',        # or cadata, or capath.
        certfile='cert.pem',
        pkeyfile='pkey.pem',
        password='...')         # for the private key, if used.
```

Note: `cadump` (output) can be used together with `cafile`/`cadata` (input), though it will be the exact copy of the CA and does not add any benefit.

As a last resort, if SSL is still a problem, it can be disabled and an insecure HTTP server can be used. This does not work with Kubernetes but can be used for direct access during development; it is also used by some tunnels that do not support HTTPS tunnelling (or require paid subscriptions):

```
@kopf.on.startup()
def config(settings: kopf.OperatorSettings, **_):
    settings.admission.server = kopf.WebhookServer(insecure=True)
```

25.12 Custom servers/tunnels

Operator developers can provide their custom servers and tunnels by implementing an async iterator over client configs (*kopf.WebhookClientConfig*). There are two ways to implement servers/tunnels.

One is a simple but non-configurable coroutine:

```
async def mytunnel(fn: kopf.WebhookFn) -> AsyncIterator[kopf.WebhookClientConfig]:
    ...
    yield client_config
    await asyncio.Event().wait()

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.admission.server = mytunnel  # no arguments!
```

Another one is a slightly more complex but configurable class:

```
class MyTunnel:
    async def __call__(self, fn: kopf.WebhookFn) -> AsyncIterator[kopf.
↪WebhookClientConfig]:
        ...
        yield client_config
        await asyncio.Event().wait()

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.admission.server = MyTunnel()  # arguments are possible.
```

The iterator MUST accept a positional argument of type *kopf.WebhookFn* and call it with the JSON-parsed payload when a review request is received; then, it MUST `await` the result and JSON-serialize it as a review response:

```
response = await fn(request)
```

Optionally (though highly recommended), several keyword arguments can be passed to extend the request data (if not passed, they all use `None` by default):

- `webhook` (`str`) – to execute only one specific handler/webhook. The id usually comes from the URL, which the framework injects automatically. It is highly recommended to provide at least this hint: otherwise, all admission handlers are executed, with mutating and validating handlers mixed, which can lead to mutating patches returned for validation requests, which in turn will fail the admission on the Kubernetes side.
- `headers` (`Mapping[str, str]`) – the HTTPS headers of the request are passed to handlers as *headers* and can be used for authentication.
- `sslpeer` (`Mapping[str, Any]`) – the SSL peer information taken from the client certificate (if provided and if verified); it is passed to handlers as *sslpeer* and can be used for authentication.

```
response = await fn(request, webhook=handler_id, headers=headers, sslpeer=sslpeer)
```

There is no guarantee on what is happening in the callback and how it works. The exact implementation can be changed in the future without warning: e.g., the framework can either invoke the admission handlers directly in the callback or queue the request for a background execution and return an awaitable future.

The iterator must yield one or more client configs. Configs are dictionaries that go to the managed webhook configurations as `.webhooks.*.clientConfig`.

Regardless of how the client config is created, the framework extends the URLs in the `url` and `service.path` fields with the handler/webhook ids, so that a URL `https://myhost/path` becomes `https://myhost/path/handler1`, `https://myhost/path/handler2`, so on.

Remember: Kubernetes prohibits using query parameters and fragments in the URLs.

In most cases, only one yielded config is enough if the server is going to serve the requests at the same endpoint. In rare cases when the endpoint changes over time (e.g. for dynamic tunnels), the server/tunnel should yield a new config every time the endpoint changes, and the webhook manager will reconfigure all managed webhooks accordingly.

The server/tunnel must hold control by running the server or by sleeping. To sleep forever, use `await asyncio.Event().wait()`. If the server/tunnel exits unexpectedly, this causes the whole operator to exit.

If the goal is to implement a tunnel only, but not a custom webhook server, it is highly advised to inherit from or directly use *kopf.WebhookServer* to run a locally listening endpoint. This server implements all URL parsing and request handling logic well-aligned with the rest of the framework:

```
# Inheritance:
class MyTunnel1(kopf.WebhookServer):
    async def __call__(self, fn: kopf.WebhookFn) -> AsyncIterator[kopf.
↪WebhookClientConfig]:
        ...
        for client_config in super().__call__(fn):
            ...  # renew a tunnel, adjust the config
            yield client_config

# Composition:
class MyTunnel2:
    async def __call__(self, fn: kopf.WebhookFn) -> AsyncIterator[kopf.
↪WebhookClientConfig]:
        server = kopf.WebhookServer(...)
        for client_config in server(fn):
            ...  # renew a tunnel, adjust the config
            yield client_config
```

25.13 System resource cleanup

It is advised that custom servers/tunnels cleanup the system resources they allocate at runtime. The easiest way is the `try-finally` block – the cleanup will happen on the garbage collection of the generator object (beware: it can be postponed in some environments, e.g. in PyPy).

For explicit cleanup of system resources, the servers/tunnels can implement the asynchronous context manager protocol:

```
class MyServer:
    def __init__(self):
        super().__init__()
        self._resource = None

    async def __aenter__(self) -> "MyServer":
        self._resource = PotentiallyLeakableResource()
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb) -> bool:
        self._resource.cleanup()
```

(continues on next page)

(continued from previous page)

```
        self._resource = None

    async def __call__(self, fn: kopf.WebhookFn) -> AsyncIterator[kopf.
↪WebhookClientConfig]:
        for client_config in super().__call__(fn):
            yield client_config
```

The context manager should usually return `self`, but it can return a substitute webhook server/tunnel object, which will actually be used. That way, the context manager turns into a factory of webhook server(s).

Keep in mind that the webhook server/tunnel is used only once per the operator's lifetime; once it exits, the whole operator stops. It makes no practical sense in making the webhook servers/tunnels reentrant.

Note: **An implementation note:** webhook servers and tunnels provided by Kopf use a little hack to keep them usable with the simple protocol (a callable that yields the client configs) while also supporting the optional context manager protocol for system resource safety: when the context manager is exited, it force-closes the generators that yield the client configs as if they were garbage-collected. Users' final webhook servers/tunnels do not need this level of complication.

See also:

For reference implementations of servers and tunnels, see the provided webhooks.

CHAPTER

TWENTYSIX

STARTUP

The startup handlers are slightly different from the module-level code: the actual tasks (e.g. API calls for resource watching) are not started until all the startup handlers succeed.

The handlers run inside of the operator's event loop, so they can initialise the loop-bound variables – which is impossible in the module-level code:

```
import asyncio
import kopf

LOCK: asyncio.Lock

@kopf.on.startup()
async def startup_fn(logger, **kwargs):
    global LOCK
    LOCK = asyncio.Lock()  # uses the running asyncio loop by default
```

If any of the startup handlers fail, the operator fails to start without making any external API calls.

Note: If the operator is running in a Kubernetes cluster, there can be timeouts set for liveness/readiness checks of a pod.

If the startup takes too long in total (e.g. due to retries), the pod can be killed by Kubernetes as not responding to the probes.

Either design the startup activities to be as fast as possible, or configure the liveness/readiness probes accordingly.

Kopf itself does not set any implicit timeouts for the startup activity, and it can continue forever (unless explicitly limited).

CHAPTER

TWENTYSEVEN

SHUTDOWN

The cleanup handlers are executed when the operator exits either by a signal (e.g. SIGTERM) or by catching an exception, or by raising the stop-flag, or by cancelling the operator's task (for *embedded operators*):

```
import kopf

@kopf.on.cleanup()
async def cleanup_fn(logger, **kwargs):
    pass
```

The cleanup handlers are not guaranteed to be fully executed if they take too long – due to a limited graceful period or non-graceful termination.

Similarly, the clean up handlers are not executed if the operator is force-killed with no possibility to react (e.g. by SIGKILL).

Note: If the operator is running in a Kubernetes cluster, there can be timeouts set for graceful termination of a pod (`terminationGracePeriodSeconds`, the default is 30 seconds).

If the cleanup takes longer than that in total (e.g. due to retries), the activity will not be finished in full, as the pod will be SIGKILL'ed by Kubernetes.

Either design the cleanup activities to be as fast as possible, or configure `terminationGracePeriodSeconds` accordingly.

Kopf itself does not set any implicit timeouts for the cleanup activity, and it can continue forever (unless explicitly limited).

CHAPTER

TWENTYEIGHT

HEALTH-CHECKS

Kopf provides a minimalistic HTTP server to report its health status.

28.1 Liveness endpoints

By default, no endpoint is configured, and no health is reported. To specify an endpoint to listen for probes, use *--liveness*:

```
kopf run --liveness=http://0.0.0.0:8080/healthz --verbose handlers.py
```

Currently, only HTTP is supported. Other protocols (TCP, HTTPS) can be added in the future.

28.2 Kubernetes probing

This port and path can be used in a liveness probe of the operator's deployment. If the operator does not respond for any reason, Kubernetes will restart it.

```
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      - name: the-only-one
        image: ...
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
```

See also:

Kubernetes manual on liveness and readiness probes.

See also:

Please be aware of the readiness vs. liveness probing. In the case of operators, readiness probing makes no practical sense, as operators do not serve traffic under the load balancing or with services. Liveness probing can help in disastrous cases (e.g. the operator is stuck), but will not help in case of partial failures (one of the API calls stuck). You can read more here: https://srcco.de/posts/kubernetes-liveness-probes-are-dangerous.html

Warning: Make sure that one and only one pod of an operator is running at a time, especially during the restarts — see *Deployment*.

28.3 Probe handlers

The content of the response is empty by default. It can be populated with probing handlers:

```
import datetime
import kopf
import random

@kopf.on.probe(id='now')
def get_current_timestamp(**kwargs):
    return datetime.datetime.now(datetime.timezone.utc).isoformat()

@kopf.on.probe(id='random')
def get_random_value(**kwargs):
    return random.randint(0, 1_000_000)
```

The probe handlers will be executed on the requests to the liveness URL, and cached for a reasonable time to prevent overloading by mass-requesting the status.

The handler results will be reported as the content of the liveness response:

```
$ curl http://localhost:8080/healthz
{"now": "2019-11-07T18:03:52.513803+00:00", "random": 765846}
```

Note: The liveness status report is simplistic and minimalistic at the moment. It only reports success if the health-reporting task runs at all. It can happen so that some of the operator's tasks, threads, or streams do break, freeze, or become unresponsive, while the health-reporting task continues to run. The probability of such a case is low, but not zero.

There are no checks that the operator operates anything (unless they are implemented explicitly with the probe-handlers), as there are no reliable criteria for that – total absence of handled resources or events can be an expected state of the cluster.

CHAPTER

TWENTYNINE

AUTHENTICATION

To access a Kubernetes cluster, an endpoint and some credentials are needed. They are usually taken either from the environment (environment variables), or from the `~/.kube/config` file, or from external authentication services.

Kopf provides rudimentary authentication out of the box: it can authenticate with the Kubernetes API either via the service account or raw kubeconfig data (with no additional interpretation or parsing of those).

But this can be not enough in some setups and environments. Kopf does not try to maintain all the authentication methods possible. Instead, it allows the operator developers to implement their custom authentication methods and "piggybacks" the existing Kubernetes clients.

The latter ones can implement some advanced authentication techniques, such as the temporary token retrieval via the authentication services, token rotation, etc.

29.1 Custom authentication

In most setups, the normal authentication from one of the API client libraries is enough — it works out of the box if those clients are installed (see *Piggybacking* below). Custom authentication is only needed if the normal authentication methods do not work for some reason, such as if you have a specific and unusual cluster setup (e.g. your own auth tokens).

To implement a custom authentication method, one or a few login-handlers can be added. The login handlers should either return nothing (`None`) or an instance of *kopf.ConnectionInfo*:

```
import datetime
import kopf

@kopf.on.login()
def login_fn(**kwargs):
    return kopf.ConnectionInfo(
        server='https://localhost',
        ca_path='/etc/ssl/ca.crt',
        ca_data=b'...',
        insecure=True,
        username='...',
        password='...',
        scheme='Bearer',
        token='...',
        certificate_path='~/.minikube/client.crt',
        private_key_path='~/.minikube/client.key',
        certificate_data=b'...',
        private_key_data=b'...',
```

(continues on next page)

(continued from previous page)

```
        expiration=datetime.datetime(2099, 12, 31, 23, 59, 59),
    )
```

Both TZ-naive & TZ-aware expiration times are supported. The TZ-naive timestamps are always treated as UTC.

As with any other handlers, the login handler can be async if the network communication is needed and async mode is supported:

```
import kopf

@kopf.on.login()
async def login_fn(**kwargs):
    pass
```

A *kopf.ConnectionInfo* is a container to bring the parameters necessary for making the API calls, but not the ways of retrieving them. Specifically:

- TCP server host & port.
- SSL verification/ignorance flag.
- SSL certificate authority.
- SSL client certificate and its private key.
- HTTP `Authorization: Basic username:password`.
- HTTP `Authorization: Bearer token` (or other schemes: Bearer, Digest, etc).
- URL's default namespace for the cases when this is implied.

No matter how the endpoints or credentials are retrieved, they are directly mapped to TCP/SSL/HTTPS protocols in the API clients. It is the responsibility of the authentication handlers to ensure that the values are consistent and valid (e.g. via internal verification calls). It is in theory possible to mix all authentication methods at once or to have none of them at all. If the credentials are inconsistent or invalid, there will be permanent re-authentication happening.

Multiple handlers can be declared to retrieve different credentials or the same credentials via different libraries. All of the retrieved credentials will be used in random order with no specific priority.

29.2 Piggybacking

In case no handlers are explicitly declared, Kopf attempts to authenticate with the existing Kubernetes libraries if they are installed. At the moment: pykube-ng and kubernetes. In the future, more libraries can be added for authentication piggybacking.

Note: Since `kopf>=1.29`, `pykube-ng` is not pre-installed implicitly. If needed, install it explicitly as a dependency of the operator, or via `kopf[full-auth]` (see *Installation*).

Piggybacking means that the config parsing and authentication methods of these libraries are used, and only the information needed for API calls is extracted.

If a few of the piggybacked libraries are installed, all of them will be attempted (as if multiple handlers are installed), and all the credentials will be utilised in random order.

If that is not the desired case, and only one of the libraries is needed, declare a custom login handler explicitly, and use only the preferred library by calling one of the piggybacking functions:

```
import kopf

@kopf.on.login()
def login_fn(**kwargs):
    return kopf.login_via_pykube(**kwargs)
```

Or:

```
import kopf

@kopf.on.login()
def login_fn(**kwargs):
    return kopf.login_via_client(**kwargs)
```

The same trick is also useful to limit the authentication attempts by time or by number of retries (by default, it tries forever until succeeded, returned nothing, or explicitly failed):

```
import kopf

@kopf.on.login(retries=3)
def login_fn(**kwargs):
    return kopf.login_via_pykube(**kwargs)
```

Similarly, if the libraries are installed and needed, but their credentials are not desired, the rudimentary login functions can be used directly:

```
import kopf

@kopf.on.login()
def login_fn(**kwargs):
    return kopf.login_with_service_account(**kwargs) or kopf.login_with_
↪kubeconfig(**kwargs)
```

See also:

kopf.login_via_pykube, *kopf.login_via_client*, *kopf.login_with_kubeconfig*, *kopf.login_with_service_account*.

29.3 Credentials lifecycle

Internally, all the credentials are gathered from all the active handlers (either the declared ones or all the fallback piggybacking ones) in no particular order, and are fed into a *vault*.

The Kubernetes API calls then use random credentials from that *vault*. The credentials that have reached their expiration are ignored and removed. If the API call fails with an HTTP 401 error, these credentials are marked invalid, excluded from further use, and the next random credentials are tried.

When the *vault* is fully depleted, it freezes all the API calls and triggers the login handlers for re-authentication. Only the new credentials are used. The credentials, which previously were known to be invalid, are ignored to prevent a permanent never-ending re-authentication loop.

There is no validation of credentials by making fake API calls. Instead, the real API calls validate the credentials by using them and reporting them back to the *vault* as invalid (or keeping them as valid), potentially causing new re-authentication activities.

In case the *vault* is depleted and no new credentials are provided by the login handlers, the API calls fail, and so does the operator.

This internal logic is hidden from the operator developers, but it is worth knowing how it works internally. See `Vault`.

If the expiration is intended to be often (e.g. every few minutes), you might want to disable the logging of re-authenication (whether this is a good idea or not, you decide using the information about your system):

```
import logging

logging.getLogger('kopf.activities.authentication').disabled = True
logging.getLogger('kopf._core.engines.activities').disabled = True
```

CHAPTER

THIRTY

CONFIGURATION

It is possible to fine-tune some aspects of Kopf-based operators, like timeouts, synchronous handler pool sizes, automatic Kubernetes Event creation from object-related log messages, etc.

30.1 Startup configuration

Every operator has its settings (even if there is more than one operator in the same processes, e.g. due to *Embedding*). The settings affect how the framework behaves in details.

The settings can be modified in the startup handlers (see *Startup*):

```
import kopf
import logging

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.posting.level = logging.WARNING
    settings.watching.connect_timeout = 1 * 60
    settings.watching.server_timeout = 10 * 60
```

All the settings have reasonable defaults, so the configuration should be used only for fine-tuning when and if necessary.

For more settings, see *kopf.OperatorSettings* and *settings*.

30.2 Logging formats and levels

The following log formats are supported on CLI:

- Full logs (the default) – with timestamps, log levels, and logger names:

  ```
  kopf run -v --log-format=full
  ```

  ```
  [2019-11-04 17:49:25,365] kopf.reactor.activit [INFO    ] Initial␣
  ↪authentication has been initiated.
  [2019-11-04 17:49:25,650] kopf.objects         [DEBUG   ] [default/kopf-
  ↪example-1] Resuming is in progress: ...
  ```

- Plain logs, with only the message:

```
kopf run -v --log-format=plain
```

```
Initial authentication has been initiated.
[default/kopf-example-1] Resuming is in progress: ...
```

For non-JSON logs, the object prefix can be disabled to make the logs completely flat (as in JSON logs):

```
kopf run -v --log-format=plain --no-log-prefix
```

```
Initial authentication has been initiated.
Resuming is in progress: ...
```

- JSON logs, with only the message:

```
kopf run -v --log-format=json
```

```
{"message": "Initial authentication has been initiated.", "severity": "info
↪", "timestamp": "2020-12-31T23:59:59.123456"}
{"message": "Resuming is in progress: ...", "object": {"apiVersion": "kopf.
↪dev/v1", "kind": "KopfExample", "name": "kopf-example-1", "uid": "...",
↪"namespace": "default"}, "severity": "debug", "timestamp": "2020-12-
↪31T23:59:59.123456"}
```

For JSON logs, the object reference key can be configured to match the log parsers (if used) – instead of the default `"object"`:

```
kopf run -v --log-format=json --log-refkey=k8s-obj
```

```
{"message": "Initial authentication has been initiated.", "severity": "info
↪", "timestamp": "2020-12-31T23:59:59.123456"}
{"message": "Resuming is in progress: ...", "k8s-obj": {...}, "severity":
↪"debug", "timestamp": "2020-12-31T23:59:59.123456"}
```

Note that the object prefixing is disabled for JSON logs by default, as the identifying information is available in the ref-keys. The prefixing can be explicitly re-enabled if needed:

```
kopf run -v --log-format=json --log-prefix
```

```
{"message": "Initial authentication has been initiated.", "severity": "info
↪", "timestamp": "2020-12-31T23:59:59.123456"}
{"message": "[default/kopf-example-1] Resuming is in progress: ...", "object
↪": {...}, "severity": "debug", "timestamp": "2020-12-31T23:59:59.123456"}
```

Note: Logging verbosity and formatting are only configured via CLI options, not via `settings.logging` as all other aspects of configuration. When the startup handlers happen for `settings`, it is too late: some initial messages could be already logged in the existing formats, or not logged when they should be due to verbosity/quietness levels.

30.3 Logging events

`settings.posting` allows to control which log messages should be posted as Kubernetes events. Use `logging` constants or integer values to set the level: e.g., `logging.WARNING`, `logging.ERROR`, etc. The default is `logging`.INFO.

```
import logging
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.posting.level = logging.ERROR
```

The event-posting can be disabled completely (the default is to be enabled):

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.posting.enabled = False
```

Note: These settings also affect *kopf.event* and related functions: *kopf.info*, *kopf.warn*, *kopf.exception*, etc – even if they are called explicitly in the code.

To avoid these settings having an impact on your code, post events directly with an API client library instead of the Kopf-provided toolkit.

30.4 Synchronous handlers

`settings.execution` allows setting the number of synchronous workers used by the operator for synchronous handlers, or replace the asyncio executor with another one:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.execution.max_workers = 20
```

It is possible to replace the whole asyncio executor used for synchronous handlers (see *Async/Await*).

Please note that the handlers that started in a previous executor, will be continued and finished with their original executor. This includes the startup handler itself. To avoid it, make the on-startup handler asynchronous:

```
import concurrent.futures
import kopf

@kopf.on.startup()
async def configure(settings: kopf.OperatorSettings, **_):
    settings.execution.executor = concurrent.futures.ThreadPoolExecutor()
```

The same executor is used both for regular sync handlers and for sync daemons. If you expect a large number of synchronous daemons (e.g. for large clusters), make sure to pre-scale the executor accordingly (the default in Python is 5x times the CPU cores):

```
import kopf

@kopf.on.startup()
async def configure(settings: kopf.OperatorSettings, **kwargs):
    settings.execution.max_workers = 1000
```

30.5 Networking timeouts

Timeouts can be controlled when communicating with Kubernetes API:

`settings.networking.request_timeout` (seconds) is how long a regular request should take before failing. This applies to all atomic requests – cluster scanning, resource patching, etc. – except the watch-streams. The default is 5 minutes (300 seconds).

`settings.networking.connect_timeout` (seconds) is how long a TCP handshake can take for regular requests before failing. There is no default (`None`), meaning that there is no timeout specifically for this; however, the handshake is limited by the overall time of the request.

`settings.watching.connect_timeout` (seconds) is how long a TCP handshake can take for watch-streams before failing. There is no default (`None`), which means that `settings.networking.connect_timeout` is used if set. If not set, then `settings.networking.request_timeout` is used.

Note: With the current aiohttp-based implementation, both connection timeouts correspond to `sock_connect=` timeout, not to `connect=` timeout, which would also include the time for getting a connection from the pool. Kopf uses unlimited aiohttp pools, so this should not be a problem.

`settings.watching.server_timeout` (seconds) is how long the session with a watching request will exist before closing it from the **server** side. This value is passed to the server-side in a query string, and the server decides on how to follow it. The watch-stream is then gracefully closed. The default is to use the server setup (`None`).

`settings.watching.client_timeout` (seconds) is how long the session with a watching request will exist before closing it from the **client** side. This includes establishing the connection and event streaming. The default is forever (`None`).

It makes no sense to set the client-side timeout shorter than the server-side timeout, but it is given to the developers' responsibility to decide.

The server-side timeouts are unpredictable, they can be 10 seconds or 10 minutes. Yet, it feels wrong to assume any "good" values in a framework (especially since it works without timeouts defined, just produces extra logs).

`settings.watching.reconnect_backoff` (seconds) is a backoff interval between watching requests – to prevent API flooding in case of errors or disconnects. The default is 0.1 seconds (nearly instant, but not flooding).

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.networking.connect_timeout = 10
    settings.networking.request_timeout = 60
    settings.watching.server_timeout = 10 * 60
```

30.6 Finalizers

A resource is blocked from deletion if the framework believes it is safer to do so, e.g. if non-optional deletion handlers are present or if daemons/timers are running at the moment.

For this, a finalizer is added to the object. It is removed when the framework believes it is safe to release the object for actual deletion.

The name of the finalizer can be configured:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.persistence.finalizer = 'my-operator.example.com/kopf-finalizer'
```

The default is the one that was hard-coded before: `kopf.zalando.org/KopfFinalizerMarker`.

30.7 Handling progress

To keep the handling state across multiple handling cycles, and to be resilient to errors and tolerable to restarts and downtimes, the operator keeps its state in a configured state storage. See more in *Continuity*.

To store the state only in the annotations with a preferred prefix:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.persistence.progress_storage = kopf.AnnotationsProgressStorage(prefix='my-
↪op.example.com')
```

To store the state only in the status or any other field:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.persistence.progress_storage = kopf.StatusProgressStorage(field='status.my-
↪operator')
```

To store in multiple places (stored in sync, but the first found state will be used when fetching, i.e. the first storage has precedence):

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.persistence.progress_storage = kopf.MultiProgressStorage([
        kopf.AnnotationsProgressStorage(prefix='my-op.example.com'),
        kopf.StatusProgressStorage(field='status.my-operator'),
    ])
```

The default storage is at both annotations and status, with annotations having precedence over the status (this is done as a transitioning solution from status-only storage in the past to annotations-only storage in the future). The annotations are `kopf.zalando.org/{id}`, the status fields are `status.kopf.progress.{id}`. It is an equivalent of:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.persistence.progress_storage = kopf.SmartProgressStorage()
```

It is also possible to implement custom state storage instead of storing the state directly in the resource's fields – e.g., in external databases. For this, inherit from *kopf.ProgressStorage* and implement its abstract methods (`fetch()`, `store()`, `purge()`, optionally `flush()`).

Note: The legacy behavior is an equivalent of `kopf.StatusProgressStorage(field='status.kopf.progress')`.

Starting with Kubernetes 1.16, both custom and built-in resources have strict structural schemas with the pruning of unknown fields (more information is in Future of CRDs: Structural Schemas).

Long story short, unknown fields are silently pruned by Kubernetes API. As a result, Kopf's status storage will not be able to store anything in the resource, as it will be instantly lost. (See #321.)

To quickly fix this for custom resources, modify their definitions with `x-kubernetes-preserve-unknown-fields: true`. For example:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
spec:
  scope: ...
  group: ...
  names: ...
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          x-kubernetes-preserve-unknown-fields: true
```

See a more verbose example in `examples/crd.yaml`.

For built-in resources, such as pods, namespaces, etc, the schemas cannot be modified, so a full switch to annotations storage is advised.

The new default "smart" storage is supposed to ensure a smooth upgrade of Kopf-based operators to the new state location without special upgrade actions or conversions needed.

30.8 Change detection

For change-detecting handlers, Kopf keeps the last handled configuration – i.e. the last state that has been successfully handled. New changes are compared against the last handled configuration, and a diff list is formed.

The last-handled configuration is also used to detect if there were any essential changes at all – i.e. not just the system or status fields.

The last-handled configuration storage can be configured with `settings.persistence.diffbase_storage`. The default is an equivalent of:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.persistence.diffbase_storage = kopf.AnnotationsDiffBaseStorage(
        prefix='kopf.zalando.org',
        key='last-handled-configuration',
    )
```

The stored content is a JSON-serialised essence of the object (i.e., only the important fields, with system fields and status stanza removed).

It is generally not a good idea to override this store unless multiple Kopf-based operators must handle the same resources, and they should not collide with each other. In that case, they must take different names.

30.9 Storage transition

Warning: Changing a storage method for an existing operator with existing resources is dangerous: the operator will consider all those resources as not handled yet (due to absence of a diff-base key) or will loose their progress state (if some handlers are retried or slow). The operator will start handling each of them again – which can lead to duplicated children or other side-effects.

To ensure a smooth transition, use a composite multi-storage, with the new storage as a first child, and the old storage as the second child (both are used for writing, the first found value is used for reading).

For example, to eventually switch from Kopf's annotations to a status field for diff-base storage, apply this configuration:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.persistence.diffbase_storage = kopf.MultiDiffBaseStorage([
        kopf.StatusDiffBaseStorage(field='status.diff-base'),
        kopf.AnnotationsDiffBaseStorage(prefix='kopf.zalando.org', key='last-handled-
↪configuration'),
    ])
```

Run the operator for some time. Let all resources change or force this: e.g. by arbitrarily labelling them, so that a new diff-base is generated:

```
kubectl label kex -l somelabel=somevalue  ping=pong
```

Then, switch to the new storage alone, without the transitional setup.

30.10 Retrying of API errors

In some cases, the Kubernetes API servers might be not ready on startup or occasionally at runtime; the network might have issues too. In most cases, these issues are of temporary nature and heal themselves withing seconds.

The framework retries the TCP/SSL networking errors and the HTTP 5xx errors ("the server is wrong") — i.e. everything that is presumed to be temporary; other errors – those presumed to be permanent, including HTTP 4xx errors ("the client is wrong") – escalate immediately without retrying.

The setting `settings.networking.error_backoffs` controls for how many times and with which backoff interval (in seconds) the retries are performed.

It is a sequence of back-offs between attempts (in seconds):

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.networking.error_backoffs = [10, 20, 30]
```

Note that the number of attempts is one more than the number of back-off intervals (because the back-offs happen inbetween the attempts).

A single integer or float value means a single backoff, i.e. 2 attempts: `(1.0)` is equivalent to `(1.0,)` or `[1.0]` for convenience.

To have a uniform back-off delay D with N+1 attempts, set to `[D] * N`.

To disable retrying (on your own risk), set it to `[]` or `()`.

The default value covers roughly a minute of attempts before giving up.

Once the retries are over (if disabled, immediately on error), the API errors escalate and are then handled according to *Throttling of unexpected errors*.

This value can be an arbitrary collection or an iterable object (even infinite): only `iter()` is called on every new retrying cycle, no other protocols are required; however, make sure that it is re-iterable for multiple uses:

```
import kopf
import random

class InfiniteBackoffsWithJitter:
    def __iter__(self):
        while True:
            yield 10 + random.randint(-5, +5)

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.networking.error_backoffs = InfiniteBackoffsWithJitter()
```

Retrying an API error blocks the task or the object's worker in which the API error happens. However, other objects and tasks run normally in parallel (unless they hit the same error in the same cluster).

Every further consecutive error leads to the next, typically bigger backoff. Every success resets the backoff intervals, and it goes from the beginning on the next error.

Note: The format is the same as for `settings.batching.error_delays`. The only difference: if the API operation does not succeed by the end of the sequence, the error of the last attempt escalates instead of blocking and retrying forever with the last delay in the sequence.

See also:

These back-offs cover only the server-side and networking errors. For errors in handlers, see *Error handling*. For errors in the framework, see *Throttling of unexpected errors*.

30.11 Throttling of unexpected errors

To prevent an uncontrollable flood of activities in case of errors that prevent the resources being marked as handled, which could lead to the Kubernetes API flooding, it is possible to throttle the activities on a per-resource basis:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.batching.error_delays = [10, 20, 30]
```

In that case, all unhandled errors in the framework or in the Kubernetes API would be backed-off by 10s after the 1st error, then by 20s after the 2nd one, and then by 30s after the 3rd, 4th, 5th errors and so on. On the first success, the backoff intervals will be reset and re-used again on the next error.

Once the errors stop and the operator is back to work, it processes only the latest event seen for that malfunctioning resource (due to event batching).

The default is a sequence of Fibonacci numbers from 1 second to 10 minutes.

The back-offs are not persisted, so they are lost on the operator restarts.

These back-offs do not cover errors in the handlers – the handlers have their own per-handler back-off intervals. These back-offs are for Kopf's own errors.

To disable throttling (on your own risk), set it to `[]` or `()`. Interpret it as: no throttling delays set — no throttling sleeps done.

If needed, this value can be an arbitrary collection/iterator/object: only `iter()` is called on every new throttling cycle, no other protocols are required; but make sure that it is re-iterable for multiple uses.

CHAPTER

THIRTYONE

PEERING

All running operators communicate with each other via peering objects (additional kind of custom resources), so they know about each other.

31.1 Priorities

Each operator has a priority (the default is 0). Whenever the operator notices that other operators start with a higher priority, it pauses its operation until those operators stop working.

This is done to prevent collisions of multiple operators handling the same objects. If two operators runs with the same priority all operators issue a warning and freeze, so that the cluster becomes not served anymore.

To set the operator's priority, use *--priority*:

```
kopf run --priority=100 ...
```

Or:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.peering.priority = 100
```

As a shortcut, there is a *--dev* option, which sets the priority to 666, and is intended for the development mode.

31.2 Scopes

There are two types of custom resources used for peering:

- `ClusterKopfPeering` for the cluster-scoped operators.
- `KopfPeering` for the namespace-scoped operators.

Kopf automatically chooses which one to use, depending on whether the operator is restricted to a namespace with *--namespace*, or it is running cluster-wide with *--all-namespaces*.

Create a peering object as needed with one of:

```yaml
apiVersion: kopf.dev/v1
kind: ClusterKopfPeering
metadata:
  name: example
```

```yaml
apiVersion: kopf.dev/v1
kind: KopfPeering
metadata:
  namespace: default
  name: example
```

Note: In `kopf<0.11` (until May 2019), `KopfPeering` was the only CRD, and it was cluster-scoped. In `kopf>=0.11,<1.29` (until Dec 2020), this mode was deprecated but supported if the old CRD existed. Since `kopf>=1.29` (Jan 2021), it is not supported anymore. To upgrade, delete and re-create the peering CRDs to the new ones.

Note: In `kopf<1.29`, all peering CRDs used the API group `kopf.zalando.org`. Since `kopf>=1.29` (Jan'2021), they belong to the API group `kopf.dev`.

At runtime, both API groups are supported. However, these resources of different API groups are mutually exclusive and cannot co-exist in the same cluster since they use the same names. Whenever possible, re-create them with the new API group after the operator/framework upgrade.

31.3 Custom peering

The operator can be instructed to use alternative peering objects:

```
kopf run --peering=example ...
kopf run --peering=example --namespace=some-ns ...
```

Or:

```python
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.peering.name = "example"
    settings.peering.mandatory = True
```

Depending on *`--namespace`* or *`--all-namespaces`*, either `ClusterKopfPeering` or `KopfPeering` will be used automatically.

If the peering object does not exist, the operator will pause at the start. Using *`--peering`* assumes that the peering is mandatory.

Please note that in the startup handler, this is not the same: the mandatory mode must be set explicitly. Otherwise, the operator will try to auto-detect the presence of the custom peering object, but will not pause if it is absent – unlike with the `--peering=` CLI option.

The operators from different peering objects do not see each other.

This is especially useful for the cluster-scoped operators for different resource kinds, which should not worry about other operators for other kinds.

31.4 Standalone mode

To prevent an operator from peering and talking to other operators, the standalone mode can be enabled:

```
kopf run --standalone ...
```

Or:

```
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.peering.standalone = True
```

In that case, the operator will not pause if other operators with the higher priority will start handling the objects, which may lead to the conflicting changes and reactions from multiple operators for the same events.

31.5 Automatic peering

If there is a peering object detected with the name `default` (either cluster-scoped or namespace-scoped), then it is used by default as the peering object.

Otherwise, Kopf will run the operator in the standalone mode.

31.6 Multi-pod operators

Usually, one and only one operator instance should be deployed for the resource. If that operator's pod dies, the handling of the resource of this type will stop until the operator's pod is restarted (and if restarted at all).

To start multiple operator pods, they must be distinctly prioritised. In that case, only one operator will be active — the one with the highest priority. All other operators will pause and wait until this operator exits. Once it dies, the second-highest priority operator will come into play. And so on.

For this, assign a monotonically growing or random priority to each operator in the deployment or replicaset:

```
kopf run --priority=$RANDOM ...
```

Or:

```
import random
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.peering.priority = random.randint(0, 32767)
```

`$RANDOM` is a feature of bash (if you use another shell, see its man page for an equivalent). It returns a random integer in the range 0..32767. With high probability, 2-3 pods will get their unique priorities.

You can also use the pod's IP address in its numeric form as the priority, or any other source of integers.

31.7 Stealth keep-alive

Every few seconds (60 by default), the operator will send a keep-alive update to the chosen peering, showing that it is still functioning. Other operators will notice that and make decisions on their pausing or resuming.

The operator also logs a keep-alive activity to its logs. This can be distracting. To disable:

```
import random
import kopf

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_):
    settings.peering.stealth = True
```

There is no equivalent CLI option for that.

Please note that it only affects logging. The keep-alive is sent anyway.

CHAPTER

THIRTYTWO

COMMAND-LINE OPTIONS

Most of the options relate to `kopf run`, though some are shared by other commands, such as `kopf freeze` and `kopf resume`.

32.1 Scripting options

`-m, --module`
: A semantical equivalent to `python -m` — which importable modules to import on startup.

32.2 Logging options

`--quiet`
: Be quiet: only show warnings and errors, but not the normal processing logs.

`--verbose`
: Show what Kopf is doing, but hide the low-level asyncio & aiohttp logs.

`--debug`
: Extremely verbose: log all the asyncio internals too, so as the API traffic.

`--log-format (plain|full|json)`
: See more in *Configuration*.

`--log-prefix, --no-log-prefix`
: Whether to prefix all object-related messages with the name of the object. By default, the prefixing is enabled.

`--log-refkey`
: For JSON logs, under which top-level key to put the object-identifying information, such as its name, namespace, etc.

32.3 Scope options

`-n, --namespace`

Serve this namespace or all namespaces mathing the pattern (or excluded from patterns). The option can be repeated multiple times.

See also:

Scopes for the pattern syntax.

`-A, --all-namespaces`

Serve the whole cluster. This is different from `--namespace *`: with `--namespace *`, the namespaces are monitored, and every resource in every namespace is watched separately, starting and stopping as needed; with `--all-namespaces`, the cluster endpoints of the Kubernetes API are used for resources, the namespaces are not monitored.

32.4 Probing options

`--liveness`

The endpoint where to serve the probes and health-checks. E.g. `http://0.0.0.0:1234/`. Only `http://` is currently supported. By default, the probing endpoint is not served.

See also:

Health-checks

32.5 Peering options

`--standalone`

Disable any peering or auto-detection of peering. Run strictly as if this is the only instance of the operator.

`--peering`

The name of the peering object to use. Depending on the operator's scope (`--all-namespaces` vs. `--namespace`, see *Scopes*), it is either `kind: KopfPeering` or `kind: ClusterKopfPeering`.

If specified, the operator will not run until that peering exists (for the namespaced operators, until it exists in each served namespace).

If not specified, the operator checks for the name "default" and uses it. If the "default" peering is absent, the operator runs in standalone mode.

`--priority`

Which priority to use for the operator. An operator with the highest priority wins the peering competitions and handlers the resources.

The default priority is `0`; `--dev` sets it to `666`.

See also:

Peering

32.6 Development mode

`--dev`

Run in the development mode. Currently, this implies `--priority=666`. Other meanings can be added in the future, such as automatic reloading of the source code.

CHAPTER

THIRTYTHREE

EVENTS

Warning: Kubernetes itself contains a terminology conflict: There are *events* when watching over the objects/resources, such as in `kubectl get pod --watch`. And there are *events* as a built-in object kind, as shown in `kubectl describe pod ...` in the "Events" section. In this documentation, they are distinguished as "watch-events" and "k8s-events". This section describes k8s-events only.

33.1 Handled objects

Kopf provides some tools to report arbitrary information for the handled objects as Kubernetes events:

```
import kopf

@kopf.on.create('kopfexamples')
def create_fn(body, **_):
    kopf.event(body,
               type='SomeType',
               reason='SomeReason',
               message='Some message')
```

The type and reason are arbitrary and can be anything. Some restrictions apply (e.g. no spaces). The message is also arbitrary free-text. However, newlines are not rendered nicely (they break the whole output of `kubectl`).

For convenience, a few shortcuts are provided to mimic the Python's `logging`:

```
import kopf

@kopf.on.create('kopfexamples')
def create_fn(body, **_):
    kopf.warn(body, reason='SomeReason', message='Some message')
    kopf.info(body, reason='SomeReason', message='Some message')
    try:
        raise RuntimeError("Exception text.")
    except:
        kopf.exception(body, reason="SomeReason", message="Some exception:")
```

These events are seen in the output of:

```
kubectl describe kopfexample kopf-example-1
```

```
...
Events:
  Type      Reason      Age   From  Message
  ----      ------      ----  ----  -------
  Normal    SomeReason  5s    kopf  Some message
  Normal    Success     5s    kopf  Handler create_fn succeeded.
  SomeType  SomeReason  6s    kopf  Some message
  Normal    Finished    5s    kopf  All handlers succeeded.
  Error     SomeReason  5s    kopf  Some exception: Exception text.
  Warning   SomeReason  5s    kopf  Some message
```

33.2 Other objects

Events can be also attached to other objects, not only those handled at the moment (and not event the children):

```
import kopf
import kubernetes

@kopf.on.create('kopfexamples')
def create_fn(name, namespace, uid, **_):

    pod = kubernetes.client.V1Pod()
    api = kubernetes.client.CoreV1Api()
    obj = api.create_namespaced_pod(namespace, pod)

    msg = f"This pod is created by KopfExample {name}"
    kopf.info(obj.to_dict(), reason='SomeReason', message=msg)
```

Note: Events are not persistent. They are usually garbage-collected after some time, e.g. one hour. All the reported information must be only for short-term use.

33.3 Events for events

As a rule of thumb, it is impossible to create "events for events".

No error will be raised. The event creation will be silently skipped.

As the primary purpose, this is done to prevent "event explosions" when handling the core v1 events, which creates new core v1 events, causing more handling, so on (similar to "fork-bombs"). Such cases are possible, for example, when using `kopf.EVERYTHING` (globally or for the v1 API), or when explicitly handling the core v1 events.

As a side-effect, "events for events" are also silenced when manually created via *kopf.event()*, *kopf.info()*, *kopf.warn()*, etc.

CHAPTER

THIRTYFOUR

HIERARCHIES

One of the most common patterns of the operators is to create children resources in the same Kubernetes cluster. Kopf provides some tools to simplify connecting these resources by manipulating their content before it is sent to the Kubernetes API.

Note: Kopf is not a Kubernetes client library. It does not provide any means to manipulate the Kubernetes resources in the cluster or to directly talk to the Kubernetes API in any other way. Use any of the existing libraries for that purpose, such as the official kubernetes client, pykorm, or pykube-ng.

In all examples below, `obj` and `objs` are either a supported object type (native or 3rd-party, see below) or a list/tuple/iterable with several objects.

34.1 Labels

To label the resources to be created, use *kopf.label()*:

```
@kopf.on.create('KopfExample')
def create_fn(**_):
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.label(objs, {'label1': 'value1', 'label2': 'value2'})
    print(objs)
    # [{'kind': 'Job',
    #   'metadata': {'labels': {'label1': 'value1', 'label2': 'value2'}}},
    #  {'kind': 'Deployment',
    #   'metadata': {'labels': {'label1': 'value1', 'label2': 'value2'}}}]
```

To label the specified resource(s) with the same labels as the resource being processed at the moment, omit the labels or set them to `None` (note, it is not the same as an empty dict `{}` – which is equivalent to doing nothing):

```
@kopf.on.create('KopfExample')
def create_fn(**_):
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.label(objs)
    print(objs)
    # [{'kind': 'Job',
    #   'metadata': {'labels': {'somelabel': 'somevalue'}}},
    #  {'kind': 'Deployment',
    #   'metadata': {'labels': {'somelabel': 'somevalue'}}}]
```

By default, if some of the requested labels already exist, they will not be overwritten. To overwrite labels, use `forced=True`:

```
@kopf.on.create('KopfExample')
def create_fn(**_):
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.label(objs, {'label1': 'value1', 'somelabel': 'not-this'}, forced=True)
    kopf.label(objs, forced=True)
    print(objs)
    # [{'kind': 'Job',
    #   'metadata': {'labels': {'label1': 'value1', 'somelabel': 'somevalue'}}},
    #  {'kind': 'Deployment',
    #   'metadata': {'labels': {'label1': 'value1', 'somelabel': 'somevalue'}}}]
```

34.2 Nested labels

For some resources, e.g. `Job` or `Deployment`, additional fields have to be modified to affect the double-nested children (`Pod` in this case).

For this, their nested fields must be mentioned in a `nested=[...]` iterable. If this is only one nested field, it can be passed directly as `nested='...'`.

If the nested structures are absent in the target resources, they are ignored and no labels are added. The labels are added only to pre-existing structures:

```
@kopf.on.create('KopfExample')
def create_fn(**_):
    objs = [{'kind': 'Job'}, {'kind': 'Deployment', 'spec': {'template': {}}}]
    kopf.label(objs, {'label1': 'value1'}, nested='spec.template')
    kopf.label(objs, nested='spec.template')
    print(objs)
    # [{'kind': 'Job',
    #   'metadata': {'labels': {'label1': 'value1', 'somelabel': 'somevalue'}}},
    #  {'kind': 'Deployment',
    #   'metadata': {'labels': {'label1': 'value1', 'somelabel': 'somevalue'}},
    #   'spec': {'template': {'metadata': {'labels': {'label1': 'value1', 'somelabel': 'somevalue'}
↪}}}}]
```

The nested structures are treated as if they were the root-level resources, i.e. they are expected to have or automatically get the `metadata` structure added.

The nested resources are labelled *in addition* to the target resources. To label only the nested resources without the root resource, pass them to the function directly (e.g., `kopf.label(obj['spec']['template'], ...)`).

34.3 Owner references

Kubernetes natively supports the owner references: a child resource can be marked as "owned" by one or more other resources (owners or parents). If the owner is deleted, its children will be deleted too, automatically, and no additional handlers are needed.

To set the ownership, use *kopf.append_owner_reference()*. To remove the ownership, use *kopf.remove_owner_reference()*:

```
kopf.append_owner_reference(objs, owner)
kopf.remove_owner_reference(objs, owner)
```

To add/remove the ownership of the requested resource(s) by the resource being processed at the moment, omit the explicit owner argument or set it to `None`:

```
@kopf.on.create('KopfExample')
def create_fn(**_):
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.append_owner_reference(objs)
    print(objs)
    # [{'kind': 'Job',
    #   'metadata': {'ownerReferences': [{'controller': True,
    #      'blockOwnerDeletion': True,
    #      'apiVersion': 'kopf.dev/v1',
    #      'kind': 'KopfExample',
    #      'name': 'kopf-example-1',
    #      'uid': '6b931859-5d50-4b5c-956b-ea2fed0d1058'}]}},
    #  {'kind': 'Deployment',
    #   'metadata': {'ownerReferences': [{'controller': True,
    #      'blockOwnerDeletion': True,
    #      'apiVersion': 'kopf.dev/v1',
    #      'kind': 'KopfExample',
    #      'name': 'kopf-example-1',
    #      'uid': '6b931859-5d50-4b5c-956b-ea2fed0d1058'}]}}]
```

To set an owner to not be a controller or not block owner deletion:

```
kopf.append_owner_reference(objs, controller=False, block_owner_deletion=False)
```

Both of the above are True by default

See also:

Cascaded deletion.

34.4 Names

It is common to name the children resources after the parent resource: either strictly as the parent, or with a random suffix.

To give the resource(s) a name, use `kopf.harmonize_naming()`. If the resource has its `metadata.name` field set, that name will be used. If it does not, the specified name will be used. It can be enforced with `forced=True`:

```
kopf.harmonize_naming(objs, 'some-name')
kopf.harmonize_naming(objs, 'some-name', forced=True)
```

By default, the specified name is used as a prefix, and a random suffix is requested from Kubernetes (via `metadata.generateName`). This is the most widely used mode with multiple children resource of the same kind. To ensure the exact name for single-child cases, pass `strict=True`:

```
kopf.harmonize_naming(objs, 'some-name', strict=True)
kopf.harmonize_naming(objs, 'some-name', strict=True, forced=True)
```

To align the name of the target resource(s) with the name of the resource being processed at the moment, omit the name or set it to `None` (both `strict=True` and `forced=True` are supported in this form too):

```
@kopf.on.create('KopfExample')
def create_fn(**_):
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.harmonize_naming(objs, forced=True, strict=True)
    print(objs)
    # [{'kind': 'Job', 'metadata': {'name': 'kopf-example-1'}},
    #  {'kind': 'Deployment', 'metadata': {'name': 'kopf-example-1'}}]
```

Alternatively, the operator can request Kubernetes to generate a name with the specified prefix and a random suffix (via `metadata.generateName`). The actual name will be known only after the resource is created:

```
@kopf.on.create('KopfExample')
def create_fn(**_):
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.harmonize_naming(objs)
    print(objs)
    # [{'kind': 'Job', 'metadata': {'generateName': 'kopf-example-1-'}},
    #  {'kind': 'Deployment', 'metadata': {'generateName': 'kopf-example-1-'}}]
```

Both ways are commonly used for parent resources that orchestrate multiple children resources of the same kind (e.g., pods in the deployment).

34.5 Namespaces

Usually, it is expected that the children resources are created in the same namespace as their parent (unless there are strong reasons to do differently).

To set the desired namespace, use `kopf.adjust_namespace()`:

```
kopf.adjust_namespace(objs, 'namespace')
```

If the namespace is already set, it will not be overwritten. To overwrite, pass `forced=True`:

```
kopf.adjust_namespace(objs, 'namespace', forced=True)
```

To align the namespace of the specified resource(s) with the namespace of the resource being processed, omit the namespace or set it to `None`:

```
@kopf.on.create('KopfExample')
def create_fn(**_):
```

(continues on next page)

(continued from previous page)

```
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.adjust_namespace(objs, forced=True)
    print(objs)
    # [{'kind': 'Job', 'metadata': {'namespace': 'default'}},
    #  {'kind': 'Deployment', 'metadata': {'namespace': 'default'}}]
```

34.6 Adopting

All of the above can be done in one call with *kopf.adopt()*; `forced`, `strict`, `nested` flags are passed to all functions that support them:

```
@kopf.on.create('KopfExample')
def create_fn(**_):
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.adopt(objs, strict=True, forced=True, nested='spec.template')
    print(objs)
    # [{'kind': 'Job',
    #   'metadata': {'ownerReferences': [{'controller': True,
    #      'blockOwnerDeletion': True,
    #      'apiVersion': 'kopf.dev/v1',
    #      'kind': 'KopfExample',
    #      'name': 'kopf-example-1',
    #      'uid': '4a15f2c2-d558-4b6e-8cf0-00585d823511'}],
    #    'name': 'kopf-example-1',
    #    'namespace': 'default',
    #    'labels': {'somelabel': 'somevalue'}}},
    #  {'kind': 'Deployment',
    #   'metadata': {'ownerReferences': [{'controller': True,
    #      'blockOwnerDeletion': True,
    #      'apiVersion': 'kopf.dev/v1',
    #      'kind': 'KopfExample',
    #      'name': 'kopf-example-1',
    #      'uid': '4a15f2c2-d558-4b6e-8cf0-00585d823511'}],
    #    'name': 'kopf-example-1',
    #    'namespace': 'default',
    #    'labels': {'somelabel': 'somevalue'}}}]
```

34.7 3rd-party libraries

All described methods support resource-related classes of selected libraries the same way as the native Python dictionaries (or any mutable mappings). Currently, that is pykube-ng (classes based on `pykube.objects.APIObject`) and kubernetes client (resource models from `kubernetes.client.models`).

```
import kopf
import pykube

@kopf.on.create('KopfExample')
def create_fn(**_):
```

(continues on next page)

(continued from previous page)

```
    api = pykube.HTTPClient(pykube.KubeConfig.from_env())
    pod = pykube.objects.Pod(api, {})
    kopf.adopt(pod)
```

```
import kopf
import kubernetes.client

@kopf.on.create('KopfExample')
def create_fn(**_):
    pod = kubernetes.client.V1Pod()
    kopf.adopt(pod)
    print(pod)
    # {'api_version': None,
    #  'kind': None,
    #  'metadata': {'annotations': None,
    #               'cluster_name': None,
    #               'creation_timestamp': None,
    #               'deletion_grace_period_seconds': None,
    #               'deletion_timestamp': None,
    #               'finalizers': None,
    #               'generate_name': 'kopf-example-1-',
    #               'generation': None,
    #               'labels': {'somelabel': 'somevalue'},
    #               'managed_fields': None,
    #               'name': None,
    #               'namespace': 'default',
    #               'owner_references': [{'api_version': 'kopf.dev/v1',
    #                                     'block_owner_deletion': True,
    #                                     'controller': True,
    #                                     'kind': 'KopfExample',
    #                                     'name': 'kopf-example-1',
    #                                     'uid': 'a114fa89-e696-4e84-9b80-b29fbccc460c'}],
    #               'resource_version': None,
    #               'self_link': None,
    #               'uid': None},
    #  'spec': None,
    #  'status': None}
```

CHAPTER

THIRTYFIVE

OPERATOR TESTING

Kopf provides some tools to test the Kopf-based operators via *kopf.testing* module (requires explicit importing).

35.1 Background runner

kopf.testing.KopfRunner runs an arbitrary operator in the background, while the original testing thread does the object manipulation and assertions:

When the `with` block exits, the operator stops, and its exceptions, exit code and output are available to the test (for additional assertions).

Listing 1: test_example_operator.py

```
import time
import subprocess
from kopf.testing import KopfRunner

def test_operator():
    with KopfRunner(['run', '-A', '--verbose', 'examples/01-minimal/example.py']) as␣
↪runner:
        # do something while the operator is running.

        subprocess.run("kubectl apply -f examples/obj.yaml", shell=True, check=True)
        time.sleep(1)  # give it some time to react and to sleep and to retry

        subprocess.run("kubectl delete -f examples/obj.yaml", shell=True, check=True)
        time.sleep(1)  # give it some time to react

    assert runner.exit_code == 0
    assert runner.exception is None
    assert 'And here we are!' in runner.stdout
    assert 'Deleted, really deleted' in runner.stdout
```

Note: The operator runs against the cluster which is currently authenticated — same as if would be executed with `kopf run`.

CHAPTER

THIRTYSIX

EMBEDDING

Kopf is designed to be embeddable into other applications, which require watching over the Kubernetes resources (custom or built-in), and handling the changes. This can be used, for example, in desktop applications or web APIs/UIs to keep the state of the cluster and its resources in memory.

36.1 Manual execution

Since Kopf is fully asynchronous, the best way to run Kopf is to provide an event-loop in a separate thread, which is dedicated to Kopf, while running the main application in the main thread:

```
import asyncio
import threading

import kopf

@kopf.on.create('kopfexamples')
def create_fn(**_):
    pass

def kopf_thread():
    asyncio.run(kopf.operator())

def main():
    thread = threading.Thread(target=kopf_thread)
    thread.start()
    # ...
    thread.join()
```

In the case of **kopf run**, the main application is Kopf itself, so its event-loop runs in the main thread.

Note: When an asyncio task runs not in the main thread, it cannot set the OS signal handlers, so a developer should implement the termination themselves (cancellation of an operator task is enough).

36.2 Manual orchestration

Alternatively, a developer can orchestrate the operator's tasks and sub-tasks themselves. The example above is an equivalent of the following:

```
def kopf_thread():
    loop = asyncio.get_event_loop_policy().get_event_loop()
    tasks = loop.run_until_complete(kopf.spawn_tasks())
    loop.run_until_complete(kopf.run_tasks(tasks, return_when=asyncio.FIRST_COMPLETED))
```

Or, if proper cancellation and termination are not expected, of the following:

```
def kopf_thread():
    loop = asyncio.get_event_loop_policy().get_event_loop()
    tasks = loop.run_until_complete(kopf.spawn_tasks())
    loop.run_until_complete(asyncio.wait(tasks))
```

In all cases, make sure that asyncio event loops are properly used. Specifically, `asyncio.run()` creates and finalises a new event loop for a single call. Several calls cannot share the coroutines and tasks. To make several calls, either create a new event loop, or get the event loop of the current asyncio _context_ (by default, of the current thread). See more on the asyncio event loops and _contexts_ in Asyncio Policies.

36.3 Custom event loops

Kopf can run in any AsyncIO-compatible event loop. For example, uvloop claims to be 2x–2.5x times faster than asyncio. To run Kopf in uvloop, call it this way:

```
import kopf
import uvloop

def main():
    loop = uvloop.EventLoopPolicy().get_event_loop()
    loop.run(kopf.operator())
```

Or this way:

```
import kopf
import uvloop

def main():
    kopf.run(loop=uvloop.EventLoopPolicy().new_event_loop())
```

Or this way:

```
import kopf
import uvloop

def main():
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    kopf.run()
```

Or any other way the event loop prescribes in its documentation.

Kopf's CLI (i.e. **kopf run**) will use uvloop by default if it is installed. To disable this implicit behaviour, either uninstall uvloop from Kopf's environment, or run Kopf explicitly from the code using the standard event loop.

For convenience, Kopf can be installed as `pip install kopf[uvloop]` to enable this mode automatically.

Kopf will never implicitly activate the custom event loops if it is called from the code, not from the CLI.

36.4 Multiple operators

Kopf can handle multiple resources at a time, so only one instance should be sufficient for most cases. However, it can be needed to run multiple isolated operators in the same process.

It should be safe to run multiple operators in multiple isolated event-loops. Despite Kopf's routines use the global state, all such a global state is stored in `contextvars` containers with values isolated per-loop and per-task.

```
import asyncio
import threading

import kopf

registry = kopf.OperatorRegistry()

@kopf.on.create('kopfexamples', registry=registry)
def create_fn(**_):
    pass

def kopf_thread():
    asyncio.run(kopf.operator(
        registry=registry,
    ))

def main():
    thread = threading.Thread(target=kopf_thread)
    thread.start()
    # ...
    thread.join()
```

Warning: It is not recommended to run Kopf in the same event-loop as other routines or applications: it considers all tasks in the event-loop as spawned by its workers and handlers, and cancels them when it exits.

There are some basic safety measures to not cancel tasks existing prior to the operator's startup, but that cannot be applied to the tasks spawned later due to asyncio implementation details.

CHAPTER

THIRTYSEVEN

DEPLOYMENT

Kopf can be executed out of the cluster, as long as the environment is authenticated to access the Kubernetes API. But normally, the operators are usually deployed directly to the clusters.

37.1 Docker image

First of all, the operator must be packaged as a docker image with Python 3.8 or newer:

Listing 1: Dockerfile

```
FROM python:3.12
RUN pip install kopf
ADD . /src
CMD kopf run /src/handlers.py --verbose
```

Build and push it to some repository of your choice. Here, we will use DockerHub (with a personal account "nolar" – replace it with your name or namespace; you may also want to add the versioning tags instead of the implied "latest"):

```
docker build -t nolar/kopf-operator .
docker push nolar/kopf-operator
```

See also:

Read DockerHub documentation for how to use it to push & pull the docker images.

37.2 Cluster deployment

The best way to deploy the operator to the cluster is via the Deployment object: in that case, it will be properly maintained alive and the versions will be properly upgraded on the re-deployments.

For this, create the deployment file:

Listing 2: deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kopfexample-operator
spec:
  replicas: 1
```

(continues on next page)

(continued from previous page)

```
  strategy:
    type: Recreate
  selector:
    matchLabels:
      application: kopfexample-operator
  template:
    metadata:
      labels:
        application: kopfexample-operator
    spec:
      serviceAccountName: kopfexample-account
      containers:
      - name: the-only-one
        image: nolar/kopf-operator
```

Please note that there is only one replica. Keep it so. If there will be two or more operators running in the cluster for the same objects, they will collide with each other and the consequences are unpredictable. In case of pod restarts, only one pod should be running at a time too: use `.spec.strategy.type=Recreate` (see the documentation).

Deploy it to the cluster:

```
kubectl apply -f deployment.yaml
```

No services or ingresses are needed (unlike in the typical web-app examples), as the operator is not listening for any incoming connections, but only makes the outcoming calls to the Kubernetes API.

37.3 RBAC

The pod where the operator runs must have the permissions to access and to manipulate the objects, both domain-specific and the built-in ones. For the example operator, those are:

- `kind: ClusterKopfPeering` for the cross-operator awareness (cluster-wide).
- `kind: KopfPeering` for the cross-operator awareness (namespace-wide).
- `kind: KopfExample` for the example operator objects.
- `kind: Pod/Job/PersistentVolumeClaim` as the children objects.
- And others as needed.

For that, the RBAC (Role-Based Access Control) could be used and attached to the operator's pod via a service account.

Here is an example of what an RBAC config should look like (remove the parts which are not needed: e.g. the cluster roles/bindings for the strictly namespace-bound operator):

Listing 3: rbac.yaml

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  namespace: "{{NAMESPACE}}"
  name: kopfexample-account
---
```

(continues on next page)

(continued from previous page)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kopfexample-role-cluster
rules:

  # Framework: knowing which other operators are running (i.e. peering).
  - apiGroups: [kopf.dev]
    resources: [clusterkopfpeerings]
    verbs: [list, watch, patch, get]

  # Framework: runtime observation of namespaces & CRDs (addition/deletion).
  - apiGroups: [apiextensions.k8s.io]
    resources: [customresourcedefinitions]
    verbs: [list, watch]
  - apiGroups: [""]
    resources: [namespaces]
    verbs: [list, watch]

  # Framework: admission webhook configuration management.
  - apiGroups: [admissionregistration.k8s.io/v1, admissionregistration.k8s.io/v1beta1]
    resources: [validatingwebhookconfigurations, mutatingwebhookconfigurations]
    verbs: [create, patch]

  # Application: read-only access for watching cluster-wide.
  - apiGroups: [kopf.dev]
    resources: [kopfexamples]
    verbs: [list, watch]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: "{{NAMESPACE}}"
  name: kopfexample-role-namespaced
rules:

  # Framework: knowing which other operators are running (i.e. peering).
  - apiGroups: [kopf.dev]
    resources: [kopfpeerings]
    verbs: [list, watch, patch, get]

  # Framework: posting the events about the handlers progress/errors.
  - apiGroups: [""]
    resources: [events]
    verbs: [create]

  # Application: watching & handling for the custom resource we declare.
  - apiGroups: [kopf.dev]
    resources: [kopfexamples]
    verbs: [list, watch, patch]

  # Application: other resources it produces and manipulates.
```

(continues on next page)

(continued from previous page)

```
  # Here, we create Jobs+PVCs+Pods, but we do not patch/update/delete them ever.
  - apiGroups: [batch, extensions]
    resources: [jobs]
    verbs: [create]
  - apiGroups: [""]
    resources: [pods, persistentvolumeclaims]
    verbs: [create]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kopfexample-rolebinding-cluster
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kopfexample-role-cluster
subjects:
  - kind: ServiceAccount
    name: kopfexample-account
    namespace: "{{NAMESPACE}}"
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: "{{NAMESPACE}}"
  name: kopfexample-rolebinding-namespaced
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: kopfexample-role-namespaced
subjects:
  - kind: ServiceAccount
    name: kopfexample-account
```

And the created service account is attached to the pods as follows:

Listing 4: deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      serviceAccountName: kopfexample-account
      containers:
      - name: the-only-one
        image: nolar/kopf-operator
```

Please note that the service accounts are always namespace-scoped. There are no cluster-wide service accounts. They must be created in the same namespace as the operator is going to run in (even if it is going to serve the whole cluster).

CHAPTER

THIRTYEIGHT

CONTINUITY

38.1 Persistence

Kopf does not have any database. It stores all the information directly on the objects in the Kubernetes cluster (which means `etcd` usually). All information is retrieved and stored via the Kubernetes API.

Specifically:

- The cross-operator exchange is performed via peering objects of type `KopfPeering` or `ClusterKopfPeering` (API versions: either `kopf.dev/v1` or `zalando.org/v1`). See *Peering* for more info.
- The last handled state of the object is stored in `metadata.annotations` (the `kopf.zalando.org/last-handled-configuration` annotation). It is used to calculate diffs upon changes.
- The handlers' state (failures, successes, retries, delays) is stored in either `metadata.annotations` (`kopf.zalando.org/{id}` keys), or in `status.kopf.progress.{id}`, where `{id}` is the handler's id.

The persistent state locations can be configured to use different keys, thus allowing multiple independent operators to handle the same resources without overlapping with each other. The above-mentioned keys are the defaults. See how to configure the stores in *Configuration* (at *Handling progress*, *Change detection*).

38.2 Restarts

It is safe to kill the operator's pod (or process) and allow it to restart.

The handlers that succeeded previously will not be re-executed. The handlers that did not execute yet, or were scheduled for retrying, will be retried by a new operators pod/process from the point where the old pod/process was terminated.

Restarting an operator will only affect the handlers currently being executed in that operator at the moment of termination, as there is no record that they have succeeded.

38.3 Downtime

If the operator is down and not running, any changes to the objects are ignored and not handled. They will be handled when the operator starts: every time a Kopf-based operator starts, it lists all objects of the resource kind, and checks for their state; if the state has changed since the object was last handled (no matter how long time ago), a new handling cycle starts.

Only the last state is taken into account. All the intermediate changes are accumulated and handled together. This corresponds to Kubernetes's concept of eventual consistency and level triggering (as opposed to edge triggering).

Warning: If the operator is down, the objects may not be deleted, as they may contain the Kopf's finalizers in `metadata.finalizers`, and Kubernetes blocks the deletion until all finalizers are removed. If the operator is not running, the finalizers will never be removed. See: *kubectl freezes on object deletion* for a work-around.

CHAPTER

THIRTYNINE

IDEMPOTENCE

Kopf provides tools to make the handlers idempotent.

The *kopf.register()* function and the `kopf.subhandler()` decorator allow to schedule arbitrary sub-handlers for the execution in the current cycle.

kopf.execute() coroutine executes arbitrary sub-handlers directly in the place of invocation, and returns when all they have succeeded.

Every one of the sub-handlers is tracked by Kopf, and will not be executed twice within one handling cycle.

```
import functools
import kopf

@kopf.on.create('kopfexamples')
async def create(spec, namespace, **kwargs):
    print("Entering create()!")  # executed ~7 times.
    await kopf.execute(fns={
        'a': create_a,
        'b': create_b,
    })
    print("Leaving create()!")  # executed 1 time only.

async def create_a(retry, **kwargs):
    if retry < 2:
        raise kopf.TemporaryError("Not ready yet.", delay=10)

async def create_b(retry, **kwargs):
    if retry < 6:
        raise kopf.TemporaryError("Not ready yet.", delay=10)
```

In this example, both `create_a` & `create_b` are submitted to Kopf as the sub-handlers of `create` on every attempt to execute it. It means, every ~10 seconds until both of the sub-handlers succeed, and the main handler succeeds too.

The first one, `create_a`, will succeed on the 3rd attempt after ~20s. The second one, `create_b`, will succeed only on the 7th attempt after ~60s.

However, despite `create_a` will be submitted whenever `create` and `create_b` are retried, it will not be executed in the 20s..60s range, as it has succeeded already, and the record about this is stored on the object.

This approach can be used to perform operations, which needs protection from double-execution, such as the children object creation with randomly generated names (e.g. Pods, Jobs, PersistentVolumeClaims, etc).

See also:

Data Persistence, *Sub-handlers*.

CHAPTER

FORTY

RECONCILIATION

Reconciliation is, in plain words, bringing the *actual state* of a system to a *desired state* as expressed by the Kubernetes resources. For example, starting as many pods, as it is declared in a deployment, especially when this declaration changes due to resource updates.

Kopf is not an operator, it is a framework to make operators. Therefore, it knows nothing about the *desired state* or *actual state* (or any *state* at all).

Kopf-based operators must implement the checks and reactions to the changes, so that both states are synchronised according to the operator's concepts.

Kopf only provides a few ways and tools for achieving this easily.

40.1 Event-driven reactions

Normally, Kopf triggers the on-creation/on-update/on-deletion handlers every time anything changes on the object, as reported by Kubernetes API. It provides both the current state of the object and a diff list with the last handled state.

The event-driven approach is the best, as it saves system resources (RAM & CPU), and does not trigger any activity when it is not needed and does not consume memory for keeping the object's last known state permanently in memory.

But it is more difficult to develop, and is not suitable for some cases: e.g., when an external non-Kubernetes system is monitored via its API.

See also:

Handlers

40.2 Regularly scheduled timers

Timers are triggered on a regular schedule, regardless of whether anything changes or does not change in the resource itself. This can be used to verify both the resource's body, and the state of other related resources through API calls, and update the original resource's status/content.

See also:

Timers

40.3 Permanently running daemons

As a last resort, a developer can implement a background task, which checks the status of the system and reacts when the "actual" state diverts from the "desired" state.

See also:

Daemons

40.4 What to use when?

As a rule of thumb _(recommended, but not insisted)_, the following guidelines can be used to decide which way of reconciliation to use in which cases:

- In the first place, try the event-driven approach by watching for the children resources (those belonging to the "actual" state).

  If there are many children resources for one parent resource, store their brief statuses on the parent's `status.children.{id}` from every individual child, and react to the changes of `status.children` in the parent resource.

- If the "desired" state can be queried with blocking waits (e.g. by running a `GET` query on a remote job/task/activity via an API, which blocks until the requested condition is reached), then use daemons to poll for the status, and process it as soon as it changes.
- If the "desired" state is not Kubernetes-related, maybe it is an external system accessed by an API, or if delays in reconciliation are acceptable, then use the timers.
- Only as the last resort, use the daemons with a `while True` cycle and explicit sleep.

CHAPTER

FORTYONE

TIPS & TRICKS

41.1 Excluding handlers forever

Both successful executions and permanent errors of change-detecting handlers only exclude these handlers from the current handling cycle, which is scoped to the current change-set (i.e. one diff of an object). On the next change, the handlers will be invoked again, regardless of their previous permanent error.

The same is valid for the daemons: they will be spawned on the next operator restart (assuming that one operator process is one handling cycle for daemons).

To prevent handlers or daemons from being invoked for a specific resource ever again, even after the operator restarts, use annotations and filters (or the same for labels or arbitrary fields with `when=` callback filtering):

```
import kopf

@kopf.on.update('kopfexamples', annotations={'update-fn-never-again': kopf.ABSENT})
def update_fn(patch, **_):
    patch.metadata.annotations['update-fn-never-again'] = 'yes'
    raise kopf.PermanentError("Never call update-fn again.")

@kopf.daemon('kopfexamples', annotations={'monitor-never-again': kopf.ABSENT})
async def monitor_kex(patch, **kwargs):
    patch.metadata.annotations['monitor-never-again'] = 'yes'
```

Such a never-again exclusion might be implemented as a feature of Kopf one day, but it is not available now – if not done explicitly as shown above.

CHAPTER

FORTYTWO

TROUBLESHOOTING

42.1 `kubectl` freezes on object deletion

This can happen if the operator is down at the moment of deletion.

The operator puts the finalizers on the objects as soon as it notices them for the first time. When the objects are *requested for deletion*, Kopf calls the deletion handlers and removes the finalizers, thus releasing the object for the *actual deletion* by Kubernetes.

If the object has to be deleted without the operator starting again, you can remove the finalizers manually:

```
kubectl patch kopfexample kopf-example-1 -p '{"metadata": {"finalizers": []}}' --type␣
↪merge
```

The object will be removed by Kubernetes immediately.

Alternatively, restart the operator, and allow it to remove the finalizers.

CHAPTER

FORTYTHREE

MINIKUBE

To develop the framework and the operators in an isolated Kubernetes cluster, use minikube.

MacOS:

```
brew install minikube
brew install hyperkit

minikube start --driver=hyperkit
minikube config set driver hyperkit
```

Start the minikube cluster:

```
minikube start
minikube dashboard
```

It automatically creates and activates the kubectl context named `minikube`. If not, or if you have multiple clusters, activate it explicitly:

```
kubectl config get-contexts
kubectl config current-context
kubectl config use-context minikube
```

For the minikube cleanup (to release the CPU/RAM/disk resources):

```
minikube stop
minikube delete
```

See also:

For even more information, read the Minikube installation manual.

CHAPTER

FORTYFOUR

CONTRIBUTING

In a nutshell, to contribute, follow this scenario:

- Fork the repo in GitHub.
- Clone the fork.
- Check out a feature branch.
- **Implement the changes.** * Lint with `pre-commit run`. * Test with `pytest`.
- Sign-off your commits.
- Create a pull request.
- Ensure all required checks are passed.
- Wait for a review by the project maintainers.

44.1 Git workflow

Kopf uses Git Forking Workflow. It means, all the development should happen in the individual forks, not in the feature branches of the main repo.

The recommended setup:

- Fork a repo on GitHub and clone the fork (not the original repo).
- Configure the `upstream` remote in addition to `origin`:

```
git remote add upstream git@github.com:nolar/kopf.git
git fetch upstream
```

- Sync your `main` branch with the upstream regularly:

```
git checkout main
git pull upstream main --ff
git push origin main
```

Work in the feature branches of your fork, not in the upstream's branches:

- Create a feature branch in the fork:

```
git checkout -b feature-x
git push origin feature-x
```

- Once the feature is ready, create a pull request from your fork to the main repo.

See also:

- Overview of the Forking Workflow.
- GitHub's manual on forking
- GitHub's manual on syncing the fork

44.2 Git conventions

The more rules you have, the less they are followed.

Kopf tries to avoid any written rules and to follow human habits and intuitive expectations where possible. Therefore:

- Write clear and explanatory commit messages and PR titles. Read How to Write a Git Commit Message for examples.
- Avoid commits' or PRs' prefixes/suffixes with the issues or change types. In general, keep the git log clean – this will later go to the changelogs.
- Sign-off your commits for DCO (see below).

No more other rules.

44.3 DCO sign-off

All contributions (including pull requests) must agree to the Developer Certificate of Origin (DCO) version 1.1. This is the same one created and used by the Linux kernel developers and posted on http://developercertificate.org/.

This is a developer's certification that they have the right to submit the patch for inclusion into the project.

Simply submitting a contribution implies this agreement. However, please include a "Signed-off-by" tag in every patch (this tag is a conventional way to confirm that you agree to the DCO):

The sign-off can be either written manually or added with `git commit -s`. If you contribute often, you can automate this in Kopf's repo with a [Git hook](https://stackoverflow.com/a/46536244/857383).

44.4 Code style

Common sense is the best code formatter. Blend your code into the surrounding code style.

Kopf does not use and will never use strict code formatters (at least until they acquire common sense and context awareness). In case of doubt, adhere to PEP-8 and [Google Python Style Guide](https://google.github.io/styleguide/pyguide.html).

The line length is 100 characters for code, 80 for docstrings and RsT files. Long URLs can exceed this length.

For linting, minor code styling, import sorting, layered modules checks, run:

```
pre-commit run
```

44.5 Tests

If possible, run the unit-tests locally before submitting (this will save you some time, but is not mandatory):

```
pytest
```

If possible, run the functional tests with a realistic local cluster (for examples, with k3s/k3d on MacOS; Kind and Minikube are also fine):

```
brew install k3d
k3d cluster create
pytest --only-e2e
```

If not possible, create a PR draft instead of a PR, and check the GitHub Actions' results for unit- & functional tests, fix as needed, and promote the PR draft into a PR once everything is ready.

44.6 Reviews

If possible, refer to an issue for which the PR is created in the PR's body. You can use one of the existing or closed issues that match your topic best.

The PRs can be reviewed and commented by anyone, but can be approved only by the project maintainers.

CHAPTER

FORTYFIVE

ARCHITECTURE

45.1 Layered layout

The framework is organized into several layers, and the layers are layered too. The higher-level layers and modules can import the lower-level ones, but not vice versa. The layering is checked and enforced by import-linter.

45.1.1 Root

At the topmost level, the framework consists of cogs, core, and kits, and user-facing modules.

`kopf`, `kopf.on`, `kopf.testing` are the public interface that can be imported by operator developers. Only these public modules contain all public promises on names and signatures. Everything else is an implementation detail.

The internal modules are intentionally hidden (by underscore-naming) to protect against introducing the dependencies on the implementation details that can change without warnings.

`cogs` are utilities used throughout the framework in nearly all modules. They do not represent the main functionality of operators but are needed to make them work. Generally, the cogs are fully independent of each other and of anything in the framework — to the point that they can be extracted as separate libraries (in theory; if anyone needs it).

`core` is the main functionality used by a Kopf-based operator. It brings the operators into motion. The core is the essence of the framework, it cannot be extracted or replaced without redefining the framework.

`kits` are utilities and specialised tools provided to operator developers for some scenarios and/or settings. The framework itself does not use them.

45.1.2 Cogs

`helpers` are system-level or language-enhancing adapters. E.g., hostname identification, dynamic Python module importing, integrations with 3rd-party libraries (such as pykube-ng or the official Kubernetes Python client).

`aiokits` are asynchronous primitives and enhancements for `asyncio`, sufficiently abstracted from the framework and the Kubernetes/operator domains.

`structs` are data structures and type declarations for Kubernetes models: such as resource kinds, selectors, bodies or their parts (specs, statuses, etc), admission reviews, so on. Besides, this includes some specialised structures, such as authentication credentials – also abstracted for the framework even in case the clients and their authentication are replaced.

`configs` are the settings mostly, and everything needed to define them: e.g. persistence storage classes (for handling progress and diff bases).

`clients` are the asynchronous adapters and wrappers for the Kubernetes API. They abstract away how the framework communicates with the API to achieve its goals (such as patching a resource or watching for its changes). Currently,

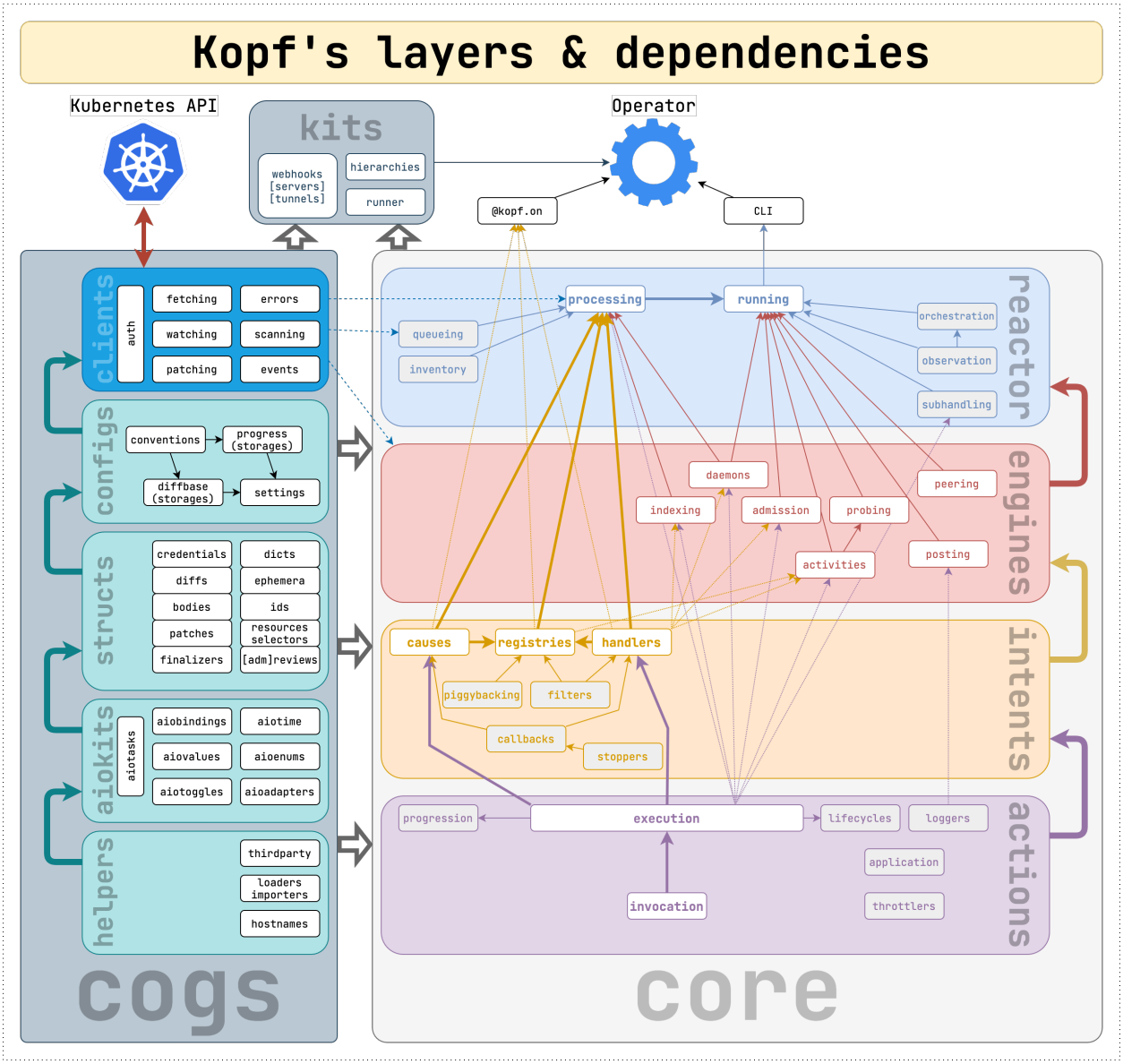

Fig. 1: Note: only the essential module dependencies are shown, not all of them. All other numerous imports are represented by cross-layer dependencies.

it is based on aiohttp; previously, it was the official Kubernetes client library and pykube-ng. Over time, the whole clients' implementation can be replaced with another one — while keeping the signatures for the rest of the framework. Only the clients are allowed to talk to the Kubernetes API.

45.1.3 Core

`actions` is the lowest level in the core (but not in the framework). It defines how the functions and handlers are invoked, which ones specifically, how their errors are handled and retried (if at all), how the function results and the patches are applied to the cluster, so on.

`intents` are mostly data structures that store the declared handlers of the operators, but also some logic to select/filter them when a reaction is needed.

`engines` are specialised aspects of the framework, i.e. its functionality. Engines are usually independent of each other (though, this is not a rule). For example, daemons and timers, validating/mutating admission requests, in-memory indexing, operator activities (authentication, probing, etc), peering, Kubernetes `kind: Event` delayed posting, etc.

`reactor` is the topmost layer in the framework. It defines the entry points for the CLI and operator embedding (see *Embedding*) and implements the task orchestration for all the engines and internal machinery. Besides, the reactor observes the cluster for resources and namespaces, and dynamically spawns/stops the tasks to serve them.

45.1.4 Kits

`hierarchies` are helper functions to manage the hierarchies of Kubernetes objects, such as labelling them, adding/removing the owner references, name generation, so on. They support raw Python dicts so as some selected libraries: pykube-ng and the official Kubernetes client for Python (see *Hierarchies*).

`webhooks` are helper servers and tunnels to accept admission requests from a Kubernetes cluster even if running locally on a developer's machines (see *Admission control*).

`runner` is a helper to run an operator in a Python context manager, mostly useful for testing (see *Operator testing*).

CHAPTER

FORTYSIX

KOPF PACKAGE

The main Kopf module for all the exported functions & classes.

kopf.**register**(*fn*, *, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *labels=None*, *annotations=None*, *when=None*)

Register a function as a sub-handler of the currently executed handler.

Example:

```
@kopf.on.create('kopfexamples')
def create_it(spec, **kwargs):
    for task in spec.get('tasks', []):

        def create_single_task(task=task, **_):
            pass

        kopf.register(id=task, fn=create_single_task)
```

This is efficiently an equivalent for:

```
@kopf.on.create('kopfexamples')
def create_it(spec, **kwargs):
    for task in spec.get('tasks', []):

        @kopf.subhandler(id=task)
        def create_single_task(task=task, **_):
            pass
```

Return type
Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]

Parameters

- **fn** (*Callable[[...], object | None | Coroutine[None, None, object | None]]*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –

- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –

async kopf.**execute**(*, *fns=None*, *handlers=None*, *registry=None*, *lifecycle=None*, *cause=None*)

Execute the handlers in an isolated lifecycle.

This function is just a public wrapper for *execute* with multiple ways to specify the handlers: either as the raw functions, or as the pre-created handlers, or as a registry (as used in the object handling).

If no explicit functions or handlers or registry are passed, the sub-handlers of the current handler are assumed, as accumulated in the per-handler registry with `@kopf.subhandler`.

If the call to this method for the sub-handlers is not done explicitly in the handler, it is done implicitly after the handler is exited. One way or another, it is executed for the sub-handlers.

Return type
None

Parameters

- **fns** (*Iterable[Callable[[...], object | None | Coroutine[None, None, object | None]]] | None*) –
- **handlers** (*Iterable[ChangingHandler] | None*) –
- **registry** (*ChangingRegistry | None*) –
- **lifecycle** (*LifeCycleFn | None*) –
- **cause** (*Cause | None*) –

kopf.**daemon**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, *, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *initial_delay=None*, *cancellation_backoff=None*, *cancellation_timeout=None*, *cancellation_polling=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.daemon()` decorator for the background threads/tasks.

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –

- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **initial_delay** (*float | None*) –
- **cancellation_backoff** (*float | None*) –
- **cancellation_timeout** (*float | None*) –
- **cancellation_polling** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.**timer**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *interval=None*, *initial_delay=None*, *sharp=None*, *idle=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.timer()` handler for the regular events.

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –

- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **interval** (*float | None*) –
- **initial_delay** (*float | None*) –
- **sharp** (*bool | None*) –
- **idle** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.**index**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.index()` handler for the indexing callbacks.

Return type

Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –

- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (ErrorsMode | *None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (OperatorRegistry | *None*) –

kopf.**configure**(*debug=None*, *verbose=None*, *quiet=None*, *log_format=LogFormat.FULL*, *log_prefix=False*, *log_refkey=None*)

Return type
None

Parameters

- **debug** (*bool | None*) –
- **verbose** (*bool | None*) –
- **quiet** (*bool | None*) –
- **log_format** (LogFormat) –
- **log_prefix** (*bool | None*) –
- **log_refkey** (*str | None*) –

class kopf.**LogFormat**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Log formats, as specified on CLI.

PLAIN = '%(message)s'

FULL = '[%(asctime)s] %(name)-20.20s [%(levelname)-8.8s] %(message)s'

JSON = '-json-'

kopf.**login_via_pykube**(*, *logger*, ***_*)

Return type
Optional[*ConnectionInfo*]

Parameters

- **logger** (*Logger | LoggerAdapter*) –
- **_** (*Any*) –

kopf.**login_via_client**(***, *logger*, ***_*)

Return type
Optional[*ConnectionInfo*]

Parameters

- **logger** (*Logger | LoggerAdapter*) –
- **_** (*Any*) –

kopf.**login_with_kubeconfig**(***_*)

A minimalistic login handler that can get raw data from a kubeconfig file.

Authentication capabilities can be limited to keep the code short & simple. No parsing or sophisticated multi-step token retrieval is performed.

This login function is intended to make Kopf runnable in trivial cases when neither pykube-ng nor the official client library are installed.

Return type
Optional[*ConnectionInfo*]

Parameters
_ (*Any*) –

kopf.**login_with_service_account**(***_*)

A minimalistic login handler that can get raw data from a service account.

Authentication capabilities can be limited to keep the code short & simple. No parsing or sophisticated multi-step token retrieval is performed.

This login function is intended to make Kopf runnable in trivial cases when neither pykube-ng nor the official client library are installed.

Return type
Optional[*ConnectionInfo*]

Parameters
_ (*Any*) –

exception kopf.**LoginError**

Bases: Exception

Raised when the operator cannot login to the API.

class kopf.**ConnectionInfo**(*server*, *ca_path=None*, *ca_data=None*, *insecure=None*, *username=None*, *password=None*, *scheme=None*, *token=None*, *certificate_path=None*, *certificate_data=None*, *private_key_path=None*, *private_key_data=None*, *default_namespace=None*, *priority=0*, *expiration=None*)

Bases: object

A single endpoint with specific credentials and connection flags to use.

Parameters

- **server** (*str*) –
- **ca_path** (*str | None*) –

- **ca_data** (*bytes | None*) –
- **insecure** (*bool | None*) –
- **username** (*str | None*) –
- **password** (*str | None*) –
- **scheme** (*str | None*) –
- **token** (*str | None*) –
- **certificate_path** (*str | None*) –
- **certificate_data** (*bytes | None*) –
- **private_key_path** (*str | None*) –
- **private_key_data** (*bytes | None*) –
- **default_namespace** (*str | None*) –
- **priority** (*int*) –
- **expiration** (*datetime | None*) –

server: str

ca_path: Optional[str] = None

ca_data: Optional[bytes] = None

insecure: Optional[bool] = None

username: Optional[str] = None

password: Optional[str] = None

scheme: Optional[str] = None

token: Optional[str] = None

certificate_path: Optional[str] = None

certificate_data: Optional[bytes] = None

private_key_path: Optional[str] = None

private_key_data: Optional[bytes] = None

default_namespace: Optional[str] = None

priority: int = 0

expiration: Optional[datetime] = None

kopf.**event**(*objs*, ***, *type*, *reason*, *message=''*)

Return type
None

Parameters

- **objs** (Body | *Iterable*[Body]) –
- **type** (*str*) –

- **reason** (*str*) –
- **message** (*str*) –

kopf.**info**(*objs*, *, *reason*, *message=''*)

Return type
None

Parameters

- **objs** (*Body | Iterable[Body]*) –
- **reason** (*str*) –
- **message** (*str*) –

kopf.**warn**(*objs*, *, *reason*, *message=''*)

Return type
None

Parameters

- **objs** (*Body | Iterable[Body]*) –
- **reason** (*str*) –
- **message** (*str*) –

kopf.**exception**(*objs*, *, *reason=''*, *message=''*, *exc=None*)

Return type
None

Parameters

- **objs** (*Body | Iterable[Body]*) –
- **reason** (*str*) –
- **message** (*str*) –
- **exc** (*BaseException | None*) –

async kopf.**spawn_tasks**(*, *lifecycle=None*, *indexers=None*, *registry=None*, *settings=None*, *memories=None*, *insights=None*, *identity=None*, *standalone=None*, *priority=None*, *peering_name=None*, *liveness_endpoint=None*, *clusterwide=False*, *namespaces=()*, *namespace=None*, *stop_flag=None*, *ready_flag=None*, *vault=None*, *memo=None*, *_command=None*)

Spawn all the tasks needed to run the operator.

The tasks are properly inter-connected with the synchronisation primitives.

Return type
Collection[Task]

Parameters

- **lifecycle** (*LifeCycleFn | None*) –
- **indexers** (*OperatorIndexers | None*) –
- **registry** (*OperatorRegistry | None*) –
- **settings** (*OperatorSettings | None*) –

- **memories** (*ResourceMemories | None*) –
- **insights** (*Insights | None*) –
- **identity** (*Identity | None*) –
- **standalone** (*bool | None*) –
- **priority** (*int | None*) –
- **peering_name** (*str | None*) –
- **liveness_endpoint** (*str | None*) –
- **clusterwide** (*bool*) –
- **namespaces** (*Collection[str | Pattern[str]]*) –
- **namespace** (*str | Pattern[str] | None*) –
- **stop_flag** (*Future | Event | Future | Event | None*) –
- **ready_flag** (*Future | Event | Future | Event | None*) –
- **vault** (*Vault | None*) –
- **memo** (*object | None*) –
- **_command** (*Coroutine[None, None, None] | None*) –

async kopf.**run_tasks**(*root_tasks*, *, *ignored=frozenset({})*)

Orchestrate the tasks and terminate them gracefully when needed.

The root tasks are expected to run forever. Their number is limited. Once any of them exits, the whole operator and all other root tasks should exit.

The root tasks, in turn, can spawn multiple sub-tasks of various purposes. They can be awaited, monitored, or fired-and-forgot.

The hung tasks are those that were spawned during the operator runtime, and were not cancelled/exited on the root tasks termination. They are given some extra time to finish, after which they are forcely terminated too. :rtype: None

---

Note: Due to implementation details, every task created after the operator's startup is assumed to be a task or a sub-task of the operator. In the end, all tasks are forcely cancelled. Even if those tasks were created by other means. There is no way to trace who spawned what. Only the tasks that existed before the operator startup are ignored (for example, those that spawned the operator itself).

Parameters

- **root_tasks** (*Collection[Task]*) –
- **ignored** (*Collection[Task]*) –

Return type
None

async kopf.**operator**(*, *lifecycle=None*, *indexers=None*, *registry=None*, *settings=None*, *memories=None*, *insights=None*, *identity=None*, *standalone=None*, *priority=None*, *peering_name=None*, *liveness_endpoint=None*, *clusterwide=False*, *namespaces=()*, *namespace=None*, *stop_flag=None*, *ready_flag=None*, *vault=None*, *memo=None*, *_command=None*)

Run the whole operator asynchronously.

This function should be used to run an operator in an asyncio event-loop if the operator is orchestrated explicitly and manually.

It is efficiently *spawn_tasks* + *run_tasks* with some safety.

Return type
None

Parameters

- **lifecycle** (*LifeCycleFn | None*) –
- **indexers** (*OperatorIndexers | None*) –
- **registry** (*OperatorRegistry | None*) –
- **settings** (*OperatorSettings | None*) –
- **memories** (*ResourceMemories | None*) –
- **insights** (*Insights | None*) –
- **identity** (*Identity | None*) –
- **standalone** (*bool | None*) –
- **priority** (*int | None*) –
- **peering_name** (*str | None*) –
- **liveness_endpoint** (*str | None*) –
- **clusterwide** (*bool*) –
- **namespaces** (*Collection[str | Pattern[str]]*) –
- **namespace** (*str | Pattern[str] | None*) –
- **stop_flag** (*Future | Event | Future | Event | None*) –
- **ready_flag** (*Future | Event | Future | Event | None*) –
- **vault** (*Vault | None*) –
- **memo** (*object | None*) –
- **_command** (*Coroutine[None, None, None] | None*) –

kopf.**run**(*, *loop=None*, *lifecycle=None*, *indexers=None*, *registry=None*, *settings=None*, *memories=None*, *insights=None*, *identity=None*, *standalone=None*, *priority=None*, *peering_name=None*, *liveness_endpoint=None*, *clusterwide=False*, *namespaces=()*, *namespace=None*, *stop_flag=None*, *ready_flag=None*, *vault=None*, *memo=None*, *_command=None*)

Run the whole operator synchronously.

If the loop is not specified, the operator runs in the event loop of the current _context_ (by asyncio's default, the current thread). See: https://docs.python.org/3/library/asyncio-policy.html for details.

Alternatively, use `asyncio.run(kopf.operator(...))` with the same options. It will take care of a new event loop's creation and finalization for this call. See: `asyncio.run()`.

Return type
None

Parameters

- **loop** (*AbstractEventLoop | None*) –
- **lifecycle** (*LifeCycleFn | None*) –
- **indexers** (*OperatorIndexers | None*) –
- **registry** (OperatorRegistry | *None*) –
- **settings** (OperatorSettings | *None*) –
- **memories** (*ResourceMemories | None*) –
- **insights** (*Insights | None*) –
- **identity** (*Identity | None*) –
- **standalone** (*bool | None*) –
- **priority** (*int | None*) –
- **peering_name** (*str | None*) –
- **liveness_endpoint** (*str | None*) –
- **clusterwide** (*bool*) –
- **namespaces** (*Collection[str | Pattern[str]]*) –
- **namespace** (*str | Pattern[str] | None*) –
- **stop_flag** (*Future | Event | Future | Event | None*) –
- **ready_flag** (*Future | Event | Future | Event | None*) –
- **vault** (*Vault | None*) –
- **memo** (*object | None*) –
- **_command** (*Coroutine[None, None, None] | None*) –

kopf.**adopt**(*objs*, *owner=None*, ***, *forced=False*, *strict=False*, *nested=None*)

The children should be in the same namespace, named after their parent, and owned by it.

Return type
None

Parameters

- **objs** (*MutableMapping[Any, Any] | _dummy | KubernetesModel | Iterable[MutableMapping[Any, Any] | _dummy | KubernetesModel]*) –
- **owner** (Body | *None*) –
- **forced** (*bool*) –
- **strict** (*bool*) –
- **nested** (*str | Iterable[None | str | Tuple[str, ...] | List[str]] | None*) –

kopf.**label**(*objs*, *labels=_UNSET.token*, ***, *forced=False*, *nested=None*, *force=None*)

Apply the labels to the object(s).

Return type
None

Parameters

- **objs** (*MutableMapping[Any, Any] | _dummy | KubernetesModel | Iterable[MutableMapping[Any, Any] | _dummy | KubernetesModel]*) –
- **labels** (*Mapping[str, None | str] | _UNSET*) –
- **forced** (*bool*) –
- **nested** (*str | Iterable[None | str | Tuple[str, ...] | List[str]] | None*) –
- **force** (*bool | None*) –

kopf.**not_**(*fn*)

Return type
TypeVar(_FnT, Callable[..., bool], Callable[..., bool])

Parameters
fn (*_FnT*) –

kopf.**all_**(*fns*)

Return type
TypeVar(_FnT, Callable[..., bool], Callable[..., bool])

Parameters
fns (*Collection[_FnT]*) –

kopf.**any_**(*fns*)

Return type
TypeVar(_FnT, Callable[..., bool], Callable[..., bool])

Parameters
fns (*Collection[_FnT]*) –

kopf.**none_**(*fns*)

Return type
TypeVar(_FnT, Callable[..., bool], Callable[..., bool])

Parameters
fns (*Collection[_FnT]*) –

kopf.**get_default_lifecycle**()

Return type
LifeCycleFn

kopf.**set_default_lifecycle**(*lifecycle*)

Return type
None

Parameters
lifecycle (*LifeCycleFn | None*) –

kopf.**build_object_reference**(*body*)

Construct an object reference for the events.

Keep in mind that some fields can be absent: e.g. `namespace` for cluster resources, or e.g. `apiVersion` for `kind: Node`, etc.

Return type
ObjectReference

Parameters
body (Body) –

kopf.**build_owner_reference**(*body*, *, *controller=True*, *block_owner_deletion=True*)

Construct an owner reference object for the parent-children relationships.

The structure needed to link the children objects to the current object as a parent. See https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/

Keep in mind that some fields can be absent: e.g. `namespace` for cluster resources, or e.g. `apiVersion` for `kind: Node`, etc.

Return type
OwnerReference

Parameters

- **body** (Body) –
- **controller** (*bool | None*) –
- **block_owner_deletion** (*bool | None*) –

kopf.**append_owner_reference**(*objs*, *owner=None*, *, *controller=True*, *block_owner_deletion=True*)

Append an owner reference to the resource(s), if it is not yet there.

Note: the owned objects are usually not the one being processed, so the whole body can be modified, no patches are needed.

Return type
None

Parameters

- **objs** (*MutableMapping[Any, Any] | _dummy | KubernetesModel | Iterable[MutableMapping[Any, Any] | _dummy | KubernetesModel]*) –
- **owner** (Body *| None*) –
- **controller** (*bool | None*) –
- **block_owner_deletion** (*bool | None*) –

kopf.**remove_owner_reference**(*objs*, *owner=None*)

Remove an owner reference to the resource(s), if it is there.

Note: the owned objects are usually not the one being processed, so the whole body can be modified, no patches are needed.

Return type
None

Parameters

- **objs** (*MutableMapping[Any, Any] | _dummy | KubernetesModel | Iterable[MutableMapping[Any, Any] | _dummy | KubernetesModel]*) –
- **owner** (Body *| None*) –

class kopf.**ErrorsMode**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

How arbitrary (non-temporary/non-permanent) exceptions are treated.

IGNORED = 1

TEMPORARY = 2

PERMANENT = 3

exception kopf.**AdmissionError**(*message=''*, *code=500*)

Bases: *PermanentError*

Raised by admission handlers when an API operation under check is bad.

An admission error behaves the same as *kopf.PermanentError*, but provides admission-specific payload for the response: a message & a numeric code.

This error type is preferred when selecting only one error to report back to apiservers as the admission review result – in case multiple handlers are called in one admission request, i.e. when the webhook endpoints are not mapped to the handler ids (e.g. when configured manually).

Parameters

- **message** (*str | None*) –
- **code** (*int | None*) –

Return type
None

class kopf.**WebhookClientConfigService**

Bases: TypedDict

namespace: Optional[str]

name: Optional[str]

path: Optional[str]

port: Optional[int]

class kopf.**WebhookClientConfig**

Bases: TypedDict

A config of clients (apiservers) to access the webhooks' server (operators).

This dictionary is put into managed webhook configurations "as is". The fields & type annotations are only for hinting.

Kopf additionally modifies the url and the service's path to inject handler ids as the last path component. This must be taken into account by custom webhook servers.

caBundle: Optional[str]

url: Optional[str]

service: Optional[*WebhookClientConfigService*]

class kopf.**UserInfo**

Bases: TypedDict

username: str

uid: str

groups: List[str]

class kopf.**WebhookFn**(**args*, ***kwargs*)

Bases: Protocol

A framework-provided function to call when an admission request is received.

The framework provides the actual function. Custom webhook servers must accept the function, invoke it accordingly on admission requests, wait for the admission response, serialise it and send it back. They do not implement this function. This protocol only declares the exact signature.

class kopf.**WebhookServer**(***, *addr=None*, *port=None*, *path=None*, *host=None*, *cadata=None*, *cafile=None*, *cadump=None*, *context=None*, *insecure=False*, *certfile=None*, *pkeyfile=None*, *password=None*, *extra_sans=()*, *verify_mode=None*, *verify_cafile=None*, *verify_capath=None*, *verify_cadata=None*)

Bases: WebhookContextManager

A local HTTP/HTTPS endpoint.

Currently, the server is based on `aiohttp`, but the implementation can change in the future without warning.

This server is also used by specialised tunnels when they need a local endpoint to be tunneled.

- `addr`, `port` is where to listen for connections (defaults to `localhost` and `9443`).
- `path` is the root path for a webhook server (defaults to no root path).
- `host` is an optional override of the hostname for webhook URLs; if not specified, the `addr` will be used.

Kubernetes requires HTTPS, so HTTPS is the default mode of the server. This webhook server supports SSL both for the server certificates and for client certificates (e.g., for authentication) at the same time:

- `cadata`, `cafile` is the CA bundle to be passed as a "client config" to the webhook configuration objects, to be used by clients/apiservers when talking to the webhook server; it is not used in the server itself.
- `cadump` is a path to save the resulting CA bundle to be used by clients, i.e. apiservers; it can be passed to `curl --cacert ...`; if `cafile` is provided, it contains the same content.
- `certfile`, `pkeyfile` define the server's endpoint certificate; if not specified, a self-signed certificate and CA will be generated for both `addr` & `host` as SANs (but only `host` for CommonName).
- `password` is either for decrypting the provided `pkeyfile`, or for encrypting and decrypting the generated private key.
- `extra_sans` are put into the self-signed certificate as SANs (DNS/IP) in addition to the host & addr (in case some other endpoints exist).
- `verify_mode`, `verify_cafile`, `verify_capath`, `verify_cadata` will be loaded into the SSL context for verifying the client certificates when provided and if provided by the clients, i.e. apiservers or curl; (`ssl.SSLContext.verify_mode`, `ssl.SSLContext.load_verify_locations`).
- `insecure` flag disables HTTPS and runs an HTTP webhook server. This is used in ngrok for a local endpoint, but can be used for debugging or when the certificate-generating dependencies/extras are not installed.

Parameters

- **addr** (*str | None*) –
- **port** (*int | None*) –
- **path** (*str | None*) –
- **host** (*str | None*) –
- **cadata** (*bytes | None*) –
- **cafile** (*str | PathLike | None*) –
- **cadump** (*str | PathLike | None*) –
- **context** (*SSLContext | None*) –
- **insecure** (*bool*) –
- **certfile** (*str | PathLike | None*) –
- **pkeyfile** (*str | PathLike | None*) –
- **password** (*str | None*) –
- **extra_sans** (*Iterable[str]*) –
- **verify_mode** (*VerifyMode | None*) –
- **verify_cafile** (*str | PathLike | None*) –
- **verify_capath** (*str | PathLike | None*) –
- **verify_cadata** (*str | bytes | None*) –

```
DEFAULT_HOST: Optional[str] = None
addr: Optional[str]
port: Optional[int]
path: Optional[str]
host: Optional[str]
cadata: Optional[bytes]
cafile: Union[str, PathLike, None]
cadump: Union[str, PathLike, None]
context: Optional[SSLContext]
insecure: bool
certfile: Union[str, PathLike, None]
pkeyfile: Union[str, PathLike, None]
password: Optional[str]
extra_sans: Iterable[str]
verify_mode: Optional[VerifyMode]
verify_cafile: Union[str, PathLike, None]
```

verify_capath: Union[str, PathLike, None]

verify_cadata: Union[str, bytes, None]

static build_certificate(*hostnames*, *password=None*)

Build a self-signed certificate with SANs (subject alternative names).

Returns a tuple of the certificate and its private key (PEM-formatted).

The certificate is "minimally sufficient", without much of the extra information on the subject besides its common and alternative names. However, IP addresses are properly recognised and normalised for better compatibility with strict SSL clients (like apiservers of Kubernetes). The first non-IP hostname becomes the certificate's common name – by convention, non-configurable. If no hostnames are found, the first IP address is used as a fallback. Magic IPs like 0.0.0.0 are excluded.

`certbuilder` is used as an implementation because it is lightweight: 2.9 MB vs. 8.7 MB for cryptography. Still, it is too heavy to include as a normal runtime dependency (for 8.8 MB of Kopf itself), so it is only available as the `kopf[dev]` extra for development-mode dependencies. This can change in the future if self-signed certificates become used at runtime (e.g. in production/staging environments or other real clusters).

Return type
Tuple[bytes, bytes]

Parameters

- **hostnames** (*Collection[str]*) –
- **password** (*str | None*) –

class kopf.**WebhookK3dServer**(*, *addr=None*, *port=None*, *path=None*, *host=None*, *cadata=None*, *cafile=None*, *cadump=None*, *context=None*, *insecure=False*, *certfile=None*, *pkeyfile=None*, *password=None*, *extra_sans=()*, *verify_mode=None*, *verify_cafile=None*, *verify_capath=None*, *verify_cadata=None*)

Bases: *WebhookServer*

A tunnel from inside of K3d/K3s to its host where the operator is running.

With this tunnel, a developer can develop the webhooks when fully offline, since all the traffic is local and never leaves the host machine.

The forwarding is maintained by K3d itself. This tunnel only replaces the endpoints for the Kubernetes webhook and injects an SSL certificate with proper CN/SANs — to match Kubernetes's SSL validity expectations.

Parameters

- **addr** (*str | None*) –
- **port** (*int | None*) –
- **path** (*str | None*) –
- **host** (*str | None*) –
- **cadata** (*bytes | None*) –
- **cafile** (*str | PathLike | None*) –
- **cadump** (*str | PathLike | None*) –
- **context** (*SSLContext | None*) –
- **insecure** (*bool*) –
- **certfile** (*str | PathLike | None*) –

- **pkeyfile** (*str | PathLike | None*) –
- **password** (*str | None*) –
- **extra_sans** (*Iterable[str]*) –
- **verify_mode** (*VerifyMode | None*) –
- **verify_cafile** (*str | PathLike | None*) –
- **verify_capath** (*str | PathLike | None*) –
- **verify_cadata** (*str | bytes | None*) –

DEFAULT_HOST: Optional[str] = 'host.k3d.internal'

class kopf.**WebhookMinikubeServer**(*, *addr=None*, *port=None*, *path=None*, *host=None*, *cadata=None*, *cafile=None*, *cadump=None*, *context=None*, *insecure=False*, *certfile=None*, *pkeyfile=None*, *password=None*, *extra_sans=()*, *verify_mode=None*, *verify_cafile=None*, *verify_capath=None*, *verify_cadata=None*)

Bases: *WebhookServer*

A tunnel from inside of Minikube to its host where the operator is running.

With this tunnel, a developer can develop the webhooks when fully offline, since all the traffic is local and never leaves the host machine.

The forwarding is maintained by Minikube itself. This tunnel only replaces the endpoints for the Kubernetes webhook and injects an SSL certificate with proper CN/SANs — to match Kubernetes's SSL validity expectations.

Parameters

- **addr** (*str | None*) –
- **port** (*int | None*) –
- **path** (*str | None*) –
- **host** (*str | None*) –
- **cadata** (*bytes | None*) –
- **cafile** (*str | PathLike | None*) –
- **cadump** (*str | PathLike | None*) –
- **context** (*SSLContext | None*) –
- **insecure** (*bool*) –
- **certfile** (*str | PathLike | None*) –
- **pkeyfile** (*str | PathLike | None*) –
- **password** (*str | None*) –
- **extra_sans** (*Iterable[str]*) –
- **verify_mode** (*VerifyMode | None*) –
- **verify_cafile** (*str | PathLike | None*) –
- **verify_capath** (*str | PathLike | None*) –
- **verify_cadata** (*str | bytes | None*) –

DEFAULT_HOST: Optional[str] = 'host.minikube.internal'

class kopf.WebhookNgrokTunnel(*, *addr=None*, *port=None*, *path=None*, *token=None*, *region=None*, *binary=None*)

Bases: WebhookContextManager

Tunnel admission webhook request via an external tunnel: ngrok.

addr, port, and path have the same meaning as in *kopf.WebhookServer*: where to listen for connections locally. Ngrok then tunnels this endpoint remotely with.

Mind that the ngrok webhook tunnel runs the local webhook server in an insecure (HTTP) mode. For secure (HTTPS) mode, a paid subscription and properly issued certificates are needed. This goes beyond Kopf's scope. If needed, implement your own ngrok tunnel.

Besides, ngrok tunnel does not report any CA to the webhook client configs. It is expected that the default trust chain is sufficient for ngrok's certs.

token can be used for paid subscriptions, which lifts some limitations. Otherwise, the free plan has a limit of 40 requests per minute (this should be enough for local development).

binary, if set, will use the specified ngrok binary path; otherwise, pyngrok downloads the binary at runtime (not recommended).

Warning: The public URL is not properly protected and a malicious user can send requests to a locally running operator. If the handlers only process the data and make no side effects, this should be fine.

Despite ngrok provides basic auth ("username:password"), Kubernetes does not permit this information in the URLs.

Ngrok partially "protects" the URLS by assigning them random hostnames. Additionally, you can add random paths. However, this is not "security", only a bit of safety for a short time (enough for development runs).

Parameters

- **addr** (*str | None*) –
- **port** (*int | None*) –
- **path** (*str | None*) –
- **token** (*str | None*) –
- **region** (*str | None*) –
- **binary** (*str | PathLike | None*) –

addr: Optional[str]

port: Optional[int]

path: Optional[str]

token: Optional[str]

region: Optional[str]

binary: Union[str, PathLike, None]

class kopf.**WebhookAutoServer**(*, *addr=None*, *port=None*, *path=None*, *host=None*, *cadata=None*, *cafile=None*, *cadump=None*, *context=None*, *insecure=False*, *certfile=None*, *pkeyfile=None*, *password=None*, *extra_sans=()*, *verify_mode=None*, *verify_cafile=None*, *verify_capath=None*, *verify_cadata=None*)

Bases: `ClusterDetector`, *WebhookServer*

A locally listening webserver which attempts to guess its proper hostname.

The choice is happening between supported webhook servers only (K3d/K3d and Minikube at the moment). In all other cases, a regular local server is started without hostname overrides.

If automatic tunneling is possible, consider *WebhookAutoTunnel* instead.

Parameters

- **addr** (*str | None*) –
- **port** (*int | None*) –
- **path** (*str | None*) –
- **host** (*str | None*) –
- **cadata** (*bytes | None*) –
- **cafile** (*str | PathLike | None*) –
- **cadump** (*str | PathLike | None*) –
- **context** (*SSLContext | None*) –
- **insecure** (*bool*) –
- **certfile** (*str | PathLike | None*) –
- **pkeyfile** (*str | PathLike | None*) –
- **password** (*str | None*) –
- **extra_sans** (*Iterable[str]*) –
- **verify_mode** (*VerifyMode | None*) –
- **verify_cafile** (*str | PathLike | None*) –
- **verify_capath** (*str | PathLike | None*) –
- **verify_cadata** (*str | bytes | None*) –

class kopf.**WebhookAutoTunnel**(*, *addr=None*, *port=None*, *path=None*)

Bases: `ClusterDetector`, `WebhookContextManager`

The same as *WebhookAutoServer*, but with possible tunneling.

Generally, tunneling gives more possibilities to run in any environment, but it must not happen without a permission from the developers, and is not possible if running in a completely isolated/local/CI/CD cluster. Therefore, developers should activated automatic setup explicitly.

If automatic tunneling is prohibited or impossible, use *WebhookAutoServer*.

Note: Automatic server/tunnel detection is highly limited in configuration and provides only the most common options of all servers & tunners: specifically, listening `addr:port/path`. All other options are specific to their servers/tunnels and the auto-guessing logic cannot use/accept/pass them.

Parameters

- **addr** (*str | None*) –
- **port** (*int | None*) –
- **path** (*str | None*) –

addr: Optional[str]

port: Optional[int]

path: Optional[str]

exception kopf.**PermanentError**

Bases: Exception

A fatal handler error, the retries are useless.

exception kopf.**TemporaryError**(*_TemporaryError__msg=None*, *delay=60*)

Bases: Exception

A potentially recoverable error, should be retried.

Parameters

- **_TemporaryError__msg** (*str | None*) –
- **delay** (*float | None*) –

Return type
None

exception kopf.**HandlerTimeoutError**

Bases: *PermanentError*

An error for the handler's timeout (if set).

exception kopf.**HandlerRetriesError**

Bases: *PermanentError*

An error for the handler's retries exceeded (if set).

class kopf.**OperatorRegistry**

Bases: object

A global registry is used for handling of multiple resources & activities.

It is usually populated by the `@kopf.on...` decorators, but can also be explicitly created and used in the embedded operators.

kopf.**get_default_registry**()

Get the default registry to be used by the decorators and the reactor unless the explicit registry is provided to them.

Return type
OperatorRegistry

kopf.**set_default_registry**(*registry*)

Set the default registry to be used by the decorators and the reactor unless the explicit registry is provided to them.

Return type
None

Parameters
registry (OperatorRegistry) –

class kopf.**OperatorSettings**(*process=<factory>*, *posting=<factory>*, *peering=<factory>*, *watching=<factory>*, *batching=<factory>*, *scanning=<factory>*, *admission=<factory>*, *execution=<factory>*, *background=<factory>*, *networking=<factory>*, *persistence=<factory>*)

Bases: object

Parameters

- **process** (*ProcessSettings*) –
- **posting** (*PostingSettings*) –
- **peering** (*PeeringSettings*) –
- **watching** (*WatchingSettings*) –
- **batching** (*BatchingSettings*) –
- **scanning** (*ScanningSettings*) –
- **admission** (*AdmissionSettings*) –
- **execution** (*ExecutionSettings*) –
- **background** (*BackgroundSettings*) –
- **networking** (*NetworkingSettings*) –
- **persistence** (*PersistenceSettings*) –

process: ProcessSettings

posting: PostingSettings

peering: PeeringSettings

watching: WatchingSettings

batching: BatchingSettings

scanning: ScanningSettings

admission: AdmissionSettings

execution: ExecutionSettings

background: BackgroundSettings

networking: NetworkingSettings

persistence: PersistenceSettings

class kopf.DiffBaseStorage

Bases: StorageKeyMarkingConvention, StorageStanzaCleaner

Store the base essence for diff calculations, i.e. last handled state.

The "essence" is a snapshot of meaningful fields, which must be tracked to identify the actual changes on the object (or absence of such).

Used in the handling routines to check if there were significant changes (i.e. not the internal and system changes, like the uids, links, etc), and to get the exact per-field diffs for the specific handler functions.

Conceptually similar to how `kubectl apply` stores the applied state on any object, and then uses that for the patch calculation: https://kubernetes.io/docs/concepts/overview/object-management-kubectl/declarative-config/

build(*, *body*, *extra_fields=None*)

Extract only the relevant fields for the state comparisons.

The framework ignores all the system fields (mostly from metadata) and the status senza completely. Except for some well-known and useful metadata, such as labels and annotations (except for sure garbage).

A special set of fields can be provided even if they are supposed to be removed. This is used, for example, for handlers which react to changes in the specific fields in the status stanza, while the rest of the status stanza is removed.

It is generally not a good idea to override this method in custom stores, unless a different definition of an object's essence is needed.

Return type
BodyEssence

Parameters

- **body** (Body) –
- **extra_fields** (*Iterable[None | str | Tuple[str, ...] | List[str]] | None*) –

abstract fetch(*, *body*)

Return type
Optional[*BodyEssence*]

Parameters
body (Body) –

abstract store(*, *body*, *patch*, *essence*)

Return type
None

Parameters

- **body** (Body) –
- **patch** (Patch) –
- **essence** (BodyEssence) –

class kopf.AnnotationsDiffBaseStorage(*, *prefix='kopf.zalando.org'*, *key='last-handled-configuration'*, *v1=True*)

Bases: StorageKeyFormingConvention, *DiffBaseStorage*

Parameters

- **prefix** (*str*) –
- **key** (*str*) –
- **v1** (*bool*) –

build(*, *body*, *extra_fields=None*)

Extract only the relevant fields for the state comparisons.

The framework ignores all the system fields (mostly from metadata) and the status senza completely. Except for some well-known and useful metadata, such as labels and annotations (except for sure garbage).

A special set of fields can be provided even if they are supposed to be removed. This is used, for example, for handlers which react to changes in the specific fields in the status stanza, while the rest of the status stanza is removed.

It is generally not a good idea to override this method in custom stores, unless a different definition of an object's essence is needed.

Return type
BodyEssence

Parameters

- **body** (Body) –
- **extra_fields** (*Iterable[None | str | Tuple[str, ...] | List[str]] | None*) –

fetch(*, *body*)

Return type
Optional[*BodyEssence*]

Parameters
body (Body) –

store(*, *body*, *patch*, *essence*)

Return type
None

Parameters

- **body** (Body) –
- **patch** (Patch) –
- **essence** (BodyEssence) –

class kopf.**StatusDiffBaseStorage**(*, *name='kopf'*, *field='status.{name}.last-handled-configuration'*)

Bases: *DiffBaseStorage*

Parameters

- **name** (*str*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –

property field: Tuple[str, ...]

build(*, *body*, *extra_fields=None*)

Extract only the relevant fields for the state comparisons.

The framework ignores all the system fields (mostly from metadata) and the status senza completely. Except for some well-known and useful metadata, such as labels and annotations (except for sure garbage).

A special set of fields can be provided even if they are supposed to be removed. This is used, for example, for handlers which react to changes in the specific fields in the status stanza, while the rest of the status stanza is removed.

It is generally not a good idea to override this method in custom stores, unless a different definition of an object's essence is needed.

Return type
BodyEssence

Parameters

- **body** (Body) –
- **extra_fields** (*Iterable[None | str | Tuple[str, ...] | List[str]] | None*) –

fetch(*, *body*)

Return type
Optional[*BodyEssence*]

Parameters
body (Body) –

store(*, *body*, *patch*, *essence*)

Return type
None

Parameters

- **body** (Body) –
- **patch** (Patch) –
- **essence** (BodyEssence) –

class kopf.**MultiDiffBaseStorage**(*storages*)

Bases: *DiffBaseStorage*

Parameters
storages (*Collection[DiffBaseStorage]*) –

build(*, *body*, *extra_fields=None*)

Extract only the relevant fields for the state comparisons.

The framework ignores all the system fields (mostly from metadata) and the status senza completely. Except for some well-known and useful metadata, such as labels and annotations (except for sure garbage).

A special set of fields can be provided even if they are supposed to be removed. This is used, for example, for handlers which react to changes in the specific fields in the status stanza, while the rest of the status stanza is removed.

It is generally not a good idea to override this method in custom stores, unless a different definition of an object's essence is needed.

Return type
BodyEssence

Parameters

- **body** (Body) –
- **extra_fields** (*Iterable[None | str | Tuple[str, ...] | List[str]] | None*) –

fetch(*, *body*)

Return type
Optional[*BodyEssence*]

Parameters
body (Body) –

store(*, *body*, *patch*, *essence*)

Return type
None

Parameters

- **body** (Body) –
- **patch** (Patch) –
- **essence** (BodyEssence) –

class kopf.**ProgressRecord**

Bases: TypedDict

A single record stored for persistence of a single handler.

started: Optional[str]

stopped: Optional[str]

delayed: Optional[str]

purpose: Optional[str]

retries: Optional[int]

success: Optional[bool]

failure: Optional[bool]

message: Optional[str]

subrefs: Optional[Collection[NewType(HandlerId, str)]]

class kopf.**ProgressStorage**

Bases: StorageStanzaCleaner

Base class and an interface for all persistent states.

The state is persisted strict per-handler, not for all handlers at once: to support overlapping operators (assuming different handler ids) storing their state on the same fields of the resource (e.g. `state.kopf`).

This also ensures that no extra logic for state merges will be needed: the handler states are atomic (i.e. state fields are not used separately) but independent: i.e. handlers should be persisted on their own, unrelated to other handlers; i.e. never combined to other atomic structures.

If combining is still needed with performance optimization in mind (e.g. for relational/transactional databases), the keys can be cached in memory for short time, and `flush()` can be overridden to actually store them.

abstract fetch(*, *key*, *body*)

Return type
Optional[*ProgressRecord*]

Parameters

- **key** (*HandlerId*) –
- **body** (Body) –

abstract store(*, *key*, *record*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*) –
- **record** (ProgressRecord) –
- **body** (Body) –
- **patch** (Patch) –

abstract purge(*, *key*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*) –
- **body** (Body) –
- **patch** (Patch) –

abstract touch(*, *body*, *patch*, *value*)

Return type
None

Parameters

- **body** (Body) –
- **patch** (Patch) –
- **value** (*str | None*) –

abstract clear(*, *essence*)

Return type
BodyEssence

Parameters
essence (BodyEssence) –

flush()

Return type
None

class kopf.**AnnotationsProgressStorage**(*, *prefix='kopf.zalando.org'*, *verbose=False*, *touch_key='touch-dummy'*, *v1=True*)

Bases: StorageKeyFormingConvention, StorageKeyMarkingConvention, *ProgressStorage*

State storage in `.metadata.annotations` with JSON-serialised content.

An example without a prefix:

An example with a prefix:

For the annotations' naming conventions, hashing, and V1 & V2 differences, see AnnotationsNamingMixin.

Parameters

- **prefix** (*str*) –
- **verbose** (*bool*) –
- **touch_key** (*str*) –
- **v1** (*bool*) –

fetch(*, *key*, *body*)

Return type
Optional[*ProgressRecord*]

Parameters

- **key** (*HandlerId*) –
- **body** (Body) –

store(*, *key*, *record*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*) –
- **record** (ProgressRecord) –
- **body** (Body) –
- **patch** (Patch) –

purge(*, *key*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*) –
- **body** (Body) –
- **patch** (Patch) –

touch(*, *body*, *patch*, *value*)

Return type
None

Parameters

- **body** (Body) –
- **patch** (Patch) –
- **value** (*str | None*) –

clear(*, *essence*)

Return type
BodyEssence

Parameters
essence (BodyEssence) –

class kopf.**StatusProgressStorage**(*, *name='kopf'*, *field='status.{name}.progress'*, *touch_field='status.{name}.dummy'*)

Bases: *ProgressStorage*

State storage in `.status` stanza with deep structure.

The structure is this:

Parameters

- **name** (*str*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **touch_field** (*None | str | Tuple[str, ...] | List[str]*) –

property field: Tuple[str, ...]

property touch_field: Tuple[str, ...]

fetch(*, *key*, *body*)

Return type
Optional[*ProgressRecord*]

Parameters

- **key** (*HandlerId*) –
- **body** (Body) –

store(*, *key*, *record*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*) –
- **record** (ProgressRecord) –
- **body** (Body) –
- **patch** (Patch) –

purge(*, *key*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*) –
- **body** (Body) –
- **patch** (Patch) –

touch(*, *body*, *patch*, *value*)

Return type
None

Parameters

- **body** (Body) –
- **patch** (Patch) –
- **value** (*str | None*) –

clear(*, *essence*)

Return type
BodyEssence

Parameters
essence (BodyEssence) –

class kopf.**MultiProgressStorage**(*storages*)

Bases: *ProgressStorage*

Parameters
storages (*Collection*[ProgressStorage]) –

fetch(*, *key*, *body*)

Return type
Optional[*ProgressRecord*]

Parameters

- **key** (*HandlerId*) –
- **body** (Body) –

store(*, *key*, *record*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*) –
- **record** (ProgressRecord) –
- **body** (Body) –
- **patch** (Patch) –

purge(*, *key*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*) –

- **body** (Body) –
- **patch** (Patch) –

touch(*, *body*, *patch*, *value*)

Return type
None

Parameters

- **body** (Body) –
- **patch** (Patch) –
- **value** (*str | None*) –

clear(*, *essence*)

Return type
BodyEssence

Parameters
essence (BodyEssence) –

class kopf.**SmartProgressStorage**(*, *name='kopf'*, *field='status.{name}.progress'*, *touch_key='touch-dummy'*, *touch_field='status.{name}.dummy'*, *prefix='kopf.zalando.org'*, *v1=True*, *verbose=False*)

Bases: *MultiProgressStorage*

Parameters

- **name** (*str*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **touch_key** (*str*) –
- **touch_field** (*None | str | Tuple[str, ...] | List[str]*) –
- **prefix** (*str*) –
- **v1** (*bool*) –
- **verbose** (*bool*) –

class kopf.**RawEvent**

Bases: TypedDict

type: Literal[None, 'ADDED', 'MODIFIED', 'DELETED']

object: *RawBody*

class kopf.**RawBody**

Bases: TypedDict

apiVersion: str

kind: str

metadata: RawMeta

spec: Mapping[str, Any]

status: Mapping[str, Any]

class kopf.**Status**(*_Status__src*)

Bases: MappingView[str, Any]

Parameters
_Status__src (Body) –

class kopf.**Spec**(*_Spec__src*)

Bases: MappingView[str, Any]

Parameters
_Spec__src (Body) –

class kopf.**Meta**(*_Meta__src*)

Bases: MappingView[str, Any]

Parameters
_Meta__src (Body) –

property labels: Mapping[str, str]

property annotations: Mapping[str, str]

property uid: str | None

property name: str | None

property namespace: NamespaceName | None

property creation_timestamp: str | None

property deletion_timestamp: str | None

class kopf.**Body**(*_Body__src*)

Bases: ReplaceableMappingView[str, Any]

Parameters
_Body__src (*Mapping[str, Any]*) –

property metadata: *Meta*

property meta: *Meta*

property spec: *Spec*

property status: *Status*

class kopf.**BodyEssence**

Bases: TypedDict

metadata: MetaEssence

spec: Mapping[str, Any]

status: Mapping[str, Any]

class kopf.**ObjectReference**

Bases: TypedDict

apiVersion: str

kind: str

namespace: Optional[str]

name: str

uid: str

class kopf.**OwnerReference**

Bases: TypedDict

controller: bool

blockOwnerDeletion: bool

apiVersion: str

kind: str

name: str

uid: str

class kopf.**Memo**

Bases: Dict[Any, Any]

A container to hold arbitrary keys-values assigned by operator developers.

It is used in the *memo* kwarg to all resource handlers, isolated per individual resource object (not the resource kind).

The values can be accessed either as dictionary keys (the memo is a `dict` under the hood) or as object attributes (except for methods of `dict`).

See more in *In-memory containers*.

```
>>> memo = Memo()
```

```
>>> memo.f1 = 100
>>> memo['f1']
... 100
```

```
>>> memo['f2'] = 200
>>> memo.f2
... 200
```

```
>>> set(memo.keys())
... {'f1', 'f2'}
```

class kopf.**Index**

Bases: Mapping[_K, *Store*[_V]], Generic[_K, _V]

A mapping of index keys to collections of values indexed under those keys.

A single index is identified by a handler id and is populated by values usually from a single indexing function (the `@kopf.index()` decorator).

Note: This class is only an abstract interface of an index. The actual implementation is in `indexing.Index`.

---

class kopf.Store

Bases: Collection[_V], Generic[_V]

A collection of all values under a single unique index key.

Multiple objects can yield the same keys, so all their values are accumulated into collections. When an object is deleted or stops matching the filters, all associated values are discarded.

The order of values is not guaranteed.

The values are not deduplicated, so duplicates are possible if multiple objects return the same values from their indexing functions.

Note: This class is only an abstract interface of an indexed store. The actual implementation is in `indexing.Store`.

---

class kopf.ObjectLogger(*, *body*, *settings*)

Bases: LoggerAdapter

A logger/adapter to carry the object identifiers for formatting.

The identifiers are then used both for formatting the per-object messages in `ObjectPrefixingFormatter`, and when posting the per-object k8s-events.

Constructed in event handling of each individual object.

The internal structure is made the same as an object reference in K8s API, but can change over time to anything needed for our internal purposes. However, as little information should be carried as possible, and the information should be protected against the object modification (e.g. in case of background posting via the queue; see `K8sPoster`).

Parameters

- **body** (Body) –
- **settings** (OperatorSettings) –

process(*msg*, *kwargs*)

Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

Return type
Tuple[str, MutableMapping[str, Any]]

Parameters

- **msg** (*str*) –
- **kwargs** (*MutableMapping[str, Any]*) –

class kopf.LocalObjectLogger(*, *body*, *settings*)

Bases: *ObjectLogger*

The same as *ObjectLogger*, but does not post the messages as k8s-events.

Used in the resource-watching handlers to log the handler's invocation successes/failures without overloading K8s with excessively many k8s-events.

This class is used internally only and is not exposed publicly in any way.

Parameters

- **body** (Body) –
- **settings** (OperatorSettings) –

log(**args*, ***kwargs*)

Delegate a log call to the underlying logger, after adding contextual information from this adapter instance.

Return type
None

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

class kopf.**Diff**(*_Diff__items*)

Bases: Sequence[*DiffItem*]

Parameters
_Diff__items (*Iterable[DiffItem]*) –

class kopf.**DiffItem**(*operation*, *field*, *old*, *new*)

Bases: NamedTuple

Parameters

- **operation** (DiffOperation) –
- **field** (*Tuple[str, ...]*) –
- **old** (*Any*) –
- **new** (*Any*) –

operation: *DiffOperation*

Alias for field number 0

field: Tuple[str, ...]

Alias for field number 1

old: Any

Alias for field number 2

new: Any

Alias for field number 3

property op: *DiffOperation*

class kopf.**DiffOperation**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: str, Enum

ADD = 'add'

CHANGE = 'change'

REMOVE = 'remove'

class kopf.Reason(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: str, Enum

CREATE = 'create'

UPDATE = 'update'

DELETE = 'delete'

RESUME = 'resume'

NOOP = 'noop'

FREE = 'free'

GONE = 'gone'

class kopf.Patch(*_Patch__src=None*, *body=None*)

Bases: Dict[str, Any]

Parameters

- **_Patch__src** (*MutableMapping[str, Any] | None*) –
- **body** (RawBody | *None*) –

property metadata: MetaPatch

property meta: MetaPatch

property spec: SpecPatch

property status: StatusPatch

as_json_patch()

Return type
List[JSONPatchItem]

class kopf.DaemonStoppingReason(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Flag

A reason or reasons of daemon being terminated.

Daemons are signalled to exit usually for two reasons: the operator itself is exiting or restarting, so all daemons of all resources must stop; or the individual resource was deleted, but the operator continues running.

No matter the reason, the daemons must exit, so one and only one stop-flag is used. Some daemons can check the reason of exiting if it is important.

There can be multiple reasons combined (in rare cases, all of them).

DONE = 1

FILTERS_MISMATCH = 2

RESOURCE_DELETED = 4

OPERATOR_PAUSING = 8

OPERATOR_EXITING = 16

DAEMON_SIGNALLED = 32

DAEMON_CANCELLED = 64

DAEMON_ABANDONED = 128

class kopf.**Resource**(*group*, *version*, *plural*, *kind=None*, *singular=None*, *shortcuts=frozenset({})*, *categories=frozenset({})*, *subresources=frozenset({})*, *namespaced=None*, *preferred=True*, *verbs=frozenset({})*)

Bases: `object`

A reference to a very specific custom or built-in resource kind.

It is used to form the K8s API URLs. Generally, K8s API only needs an API group, an API version, and a plural name of the resource. All other names are remembered to match against resource selectors, for logging, and for informational purposes.

Parameters

- **group** (*str*) –
- **version** (*str*) –
- **plural** (*str*) –
- **kind** (*str | None*) –
- **singular** (*str | None*) –
- **shortcuts** (*FrozenSet[str]*) –
- **categories** (*FrozenSet[str]*) –
- **subresources** (*FrozenSet[str]*) –
- **namespaced** (*bool | None*) –
- **preferred** (*bool*) –
- **verbs** (*FrozenSet[str]*) –

group: str

The resource's API group; e.g. `"kopf.dev"`, `"apps"`, `"batch"`. For Core v1 API resources, an empty string: `""`.

version: str

The resource's API version; e.g. `"v1"`, `"v1beta1"`, etc.

plural: str

The resource's plural name; e.g. `"pods"`, `"kopfexamples"`. It is used as an API endpoint, together with API group & version.

kind: Optional[str] = None

The resource's kind (as in YAML files); e.g. `"Pod"`, `"KopfExample"`.

singular: Optional[str] = None

The resource's singular name; e.g. `"pod"`, `"kopfexample"`.

shortcuts: FrozenSet[str] = frozenset({})

The resource's short names; e.g. `{"po"}`, `{"kex", "kexes"}`.

categories: FrozenSet[str] = frozenset({})

The resource's categories, to which the resource belongs; e.g. {"all"}.

subresources: FrozenSet[str] = frozenset({})

The resource's subresources, if defined; e.g. {"status", "scale"}.

namespaced: Optional[bool] = None

Whether the resource is namespaced (True) or cluster-scoped (False).

preferred: bool = True

Whether the resource belong to a "preferred" API version. Only "preferred" resources are served when the version is not specified.

verbs: FrozenSet[str] = frozenset({})

All available verbs for the resource, as supported by K8s API; e.g., {"list", "watch", "create", "update", "delete", "patch"}. Note that it is not the same as all verbs permitted by RBAC.

get_url(*, *server=None*, *namespace=None*, *name=None*, *subresource=None*, *params=None*)

Build a URL to be used with K8s API.

If the namespace is not set, a cluster-wide URL is returned. For cluster-scoped resources, the namespace is ignored.

If the name is not set, the URL for the resource list is returned. Otherwise (if set), the URL for the individual resource is returned.

If subresource is set, that subresource's URL is returned, regardless of whether such a subresource is known or not.

Params go to the query parameters (?param1=value1&param2=value2...).

Return type
str

Parameters

- **server** (*str | None*) –
- **namespace** (*NamespaceName | None*) –
- **name** (*str | None*) –
- **subresource** (*str | None*) –
- **params** (*Mapping[str, str] | None*) –

46.1 Submodules

46.1.1 kopf.cli module

class kopf.cli.**CLIControls**(*ready_flag=None*, *stop_flag=None*, *vault=None*, *registry=None*, *settings=None*, *loop=None*)

Bases: object

KopfRunner controls, which are impossible to pass via CLI.

Parameters

- **ready_flag** (*Future | Event | Future | Event | None*) –

- **stop_flag** (*Future | Event | Future | Event | None*) –
- **vault** (*Vault | None*) –
- **registry** (OperatorRegistry | *None*) –
- **settings** (OperatorSettings | *None*) –
- **loop** (*AbstractEventLoop | None*) –

ready_flag: Union[Future, Event, Future, Event, None] = None

stop_flag: Union[Future, Event, Future, Event, None] = None

vault: Optional[Vault] = None

registry: Optional[*OperatorRegistry*] = None

settings: Optional[*OperatorSettings*] = None

loop: Optional[AbstractEventLoop] = None

class kopf.cli.**LogFormatParamType**

Bases: Choice

convert(*value*, *param*, *ctx*)

Convert the value to the correct type. This is not called if the value is None (the missing value).

This must accept string values from the command line, as well as values that are already the correct type. It may also convert other compatible types.

The param and ctx arguments may be None in certain situations, such as when converting prompt input.

If the value cannot be converted, call fail() with a descriptive message.

Parameters

- **value** (Any) – The value to convert.
- **param** (Any) – The parameter that is using this type to convert its value. May be None.
- **ctx** (Any) – The current context that arrived at this value. May be None.

Return type
LogFormat

kopf.cli.**logging_options**(*fn*)

A decorator to configure logging in all commands the same way.

Return type
Callable[..., Any]

Parameters
fn (*Callable[[...], Any]*) –

46.1.2 kopf.on module

The decorators for the event handlers. Usually used as:

```
import kopf

@kopf.on.create('kopfexamples')
def creation_handler(**kwargs):
    pass
```

This module is a part of the framework's public interface.

kopf.on.**startup**(*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *registry=None*)

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (ErrorsMode | *None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **registry** (OperatorRegistry | *None*) –

kopf.on.**cleanup**(*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *registry=None*)

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (ErrorsMode | *None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **registry** (OperatorRegistry | *None*) –

kopf.on.**login**(*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *registry=None*)

`@kopf.on.login()` handler for custom (re-)authentication.

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (ErrorsMode | *None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **registry** (OperatorRegistry | *None*) –

kopf.on.**probe**(*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *registry=None*)

`@kopf.on.probe()` handler for arbitrary liveness metrics.

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (ErrorsMode | *None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **registry** (OperatorRegistry | *None*) –

kopf.on.**validate**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, *, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *operation=None*, *operations=None*, *subresource=None*, *persistent=None*, *side_effects=None*, *ignore_failures=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.on.validate()` handler for validating admission webhooks.

Return type
Callable[[Callable[..., Optional[Coroutine[None, None, None]]]], Callable[..., Optional[Coroutine[None, None, None]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –

- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **operation** (*Literal['CREATE', 'UPDATE', 'DELETE', 'CONNECT'] | None*) –
- **operations** (*Collection[Literal['CREATE', 'UPDATE', 'DELETE', 'CONNECT']] | None*) –
- **subresource** (*str | None*) –
- **persistent** (*bool | None*) –
- **side_effects** (*bool | None*) –
- **ignore_failures** (*bool | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**mutate**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *operation=None*, *operations=None*, *subresource=None*, *persistent=None*, *side_effects=None*, *ignore_failures=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.on.mutate()` handler for mutating admission webhooks.

Return type
Callable[[Callable[..., Optional[Coroutine[None, None, None]]]], Callable[..., Optional[Coroutine[None, None, None]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –

- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **operation** (*Literal['CREATE', 'UPDATE', 'DELETE', 'CONNECT'] | None*) –
- **operations** (*Collection[Literal['CREATE', 'UPDATE', 'DELETE', 'CONNECT']] | None*) –
- **subresource** (*str | None*) –
- **persistent** (*bool | None*) –
- **side_effects** (*bool | None*) –
- **ignore_failures** (*bool | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**resume**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *deleted=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.on.resume()` handler for the object resuming on operator (re)start.

Return type

Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –

- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **deleted** (*bool | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**create**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.on.create()` handler for the object creation.

Return type

Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –

- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**update**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *old=None*, *new=None*, *registry=None*)

`@kopf.on.update()` handler for the object update or change.

Return type

Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –

- **backoff** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **old** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **new** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**delete**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *optional=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.on.delete()` handler for the object deletion.

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **optional** (*bool | None*) –

- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**field**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *labels=None*, *annotations=None*, *when=None*, *field*, *value=None*, *old=None*, *new=None*, *registry=None*)

`@kopf.on.field()` handler for the individual field changes.

Return type

Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –

- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **old** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **new** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**index**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.index()` handler for the indexing callbacks.

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –

- **registry** (OperatorRegistry | None) –

kopf.on.**event**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.on.event()` handler for the silent spies on the events.

Return type

Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (OperatorRegistry | None) –

kopf.on.**daemon**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *initial_delay=None*, *cancellation_backoff=None*, *cancellation_timeout=None*, *cancellation_polling=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.daemon()` decorator for the background threads/tasks.

Return type

Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **initial_delay** (*float | None*) –
- **cancellation_backoff** (*float | None*) –
- **cancellation_timeout** (*float | None*) –
- **cancellation_polling** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**timer**(*__group_or_groupversion_or_name=None*, *__version_or_name=None*, *__name=None*, ***, *group=None*, *version=None*, *kind=None*, *plural=None*, *singular=None*, *shortcut=None*, *category=None*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *interval=None*, *initial_delay=None*, *sharp=None*, *idle=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *registry=None*)

`@kopf.timer()` handler for the regular events.

Return type

Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **__group_or_groupversion_or_name** (*str | Marker | None*) –
- **__version_or_name** (*str | Marker | None*) –
- **__name** (*str | Marker | None*) –
- **group** (*str | None*) –
- **version** (*str | None*) –
- **kind** (*str | None*) –
- **plural** (*str | None*) –
- **singular** (*str | None*) –
- **shortcut** (*str | None*) –
- **category** (*str | None*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **interval** (*float | None*) –
- **initial_delay** (*float | None*) –
- **sharp** (*bool | None*) –
- **idle** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **registry** (*OperatorRegistry | None*) –

kopf.on.**subhandler**(*, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *labels=None*, *annotations=None*, *when=None*, *field=None*, *value=None*, *old=None*, *new=None*)

`@kopf.subhandler()` decorator for the dynamically generated sub-handlers.

Can be used only inside of the handler function. It is efficiently a syntax sugar to look like all other handlers:

```
@kopf.on.create('kopfexamples')
def create(*, spec, **kwargs):

    for task in spec.get('tasks', []):
```

(continues on next page)

(continued from previous page)

```
    @kopf.subhandler(id=f'task_{task}')
    def create_task(*, spec, task=task, **kwargs):
        pass
```

In this example, having spec.tasks set to `[abc, def]`, this will create the following handlers: `create`, `create/task_abc`, `create/task_def`.

The parent handler is not considered as finished if there are unfinished sub-handlers left. Since the sub-handlers will be executed in the regular reactor and lifecycle, with multiple low-level events (one per iteration), the parent handler will also be executed multiple times, and is expected to produce the same (or at least predictable) set of sub-handlers. In addition, keep its logic idempotent (not failing on the repeated calls).

Note: `task=task` is needed to freeze the closure variable, so that every create function will have its own value, not the latest in the for-cycle.

Return type
Callable[[Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]], Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]]

Parameters

- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –
- **field** (*None | str | Tuple[str, ...] | List[str]*) –
- **value** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **old** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –
- **new** (*None | Any | MetaFilterToken | Callable[[...], bool]*) –

kopf.on.**register**(*fn*, *, *id=None*, *param=None*, *errors=None*, *timeout=None*, *retries=None*, *backoff=None*, *labels=None*, *annotations=None*, *when=None*)

Register a function as a sub-handler of the currently executed handler.

Example:

```
@kopf.on.create('kopfexamples')
def create_it(spec, **kwargs):
    for task in spec.get('tasks', []):
```

(continues on next page)

(continued from previous page)

```
        def create_single_task(task=task, **_):
            pass

        kopf.register(id=task, fn=create_single_task)
```

This is efficiently an equivalent for:

```
@kopf.on.create('kopfexamples')
def create_it(spec, **kwargs):
    for task in spec.get('tasks', []):

        @kopf.subhandler(id=task)
        def create_single_task(task=task, **_):
            pass
```

Return type
Callable[..., Union[object, None, Coroutine[None, None, Optional[object]]]]

Parameters

- **fn** (*Callable[[...], object | None | Coroutine[None, None, object | None]]*) –
- **id** (*str | None*) –
- **param** (*Any | None*) –
- **errors** (*ErrorsMode | None*) –
- **timeout** (*float | None*) –
- **retries** (*int | None*) –
- **backoff** (*float | None*) –
- **labels** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **annotations** (*Mapping[str, str | MetaFilterToken | Callable[[...], bool]] | None*) –
- **when** (*Callable[[...], bool] | None*) –

46.1.3 kopf.testing module

Helper tools to test the Kopf-based operators.

This module is a part of the framework's public interface.

class kopf.testing.**KopfRunner**(**args*, *reraise=True*, *timeout=None*, *registry=None*, *settings=None*, ***kwargs*)

Bases: _AbstractKopfRunner

A context manager to run a Kopf-based operator in parallel with the tests.

Usage:

```
from kopf.testing import KopfRunner

with KopfRunner(['run', '-A', '--verbose', 'examples/01-minimal/example.py']) as␣
↪runner:
    # do something while the operator is running.
    time.sleep(3)

assert runner.exit_code == 0
assert runner.exception is None
assert 'And here we are!' in runner.stdout
```

All the args & kwargs are passed directly to Click's invocation method. See: `click.testing.CliRunner`. All properties proxy directly to Click's `click.testing.Result` object when it is available (i.e. after the context manager exits).

CLI commands have to be invoked in parallel threads, never in processes:

First, with multiprocessing, they are unable to pickle and pass exceptions (specifically, their traceback objects) from a child thread (Kopf's CLI) to the parent thread (pytest).

Second, mocking works within one process (all threads), but not across processes — the mock's calls (counts, arrgs) are lost.

Parameters

- **args** (*Any*) –
- **reraise** (*bool*) –
- **timeout** (*float | None*) –
- **registry** (*OperatorRegistry | None*) –
- **settings** (*OperatorSettings | None*) –
- **kwargs** (*Any*) –

property future: Future

property output: str

property stdout: str

property stdout_bytes: bytes

property stderr: str

property stderr_bytes: bytes

property exit_code: int

property exception: BaseException

property exc_info: Tuple[Type[BaseException], BaseException, TracebackType]

CHAPTER

FORTYSEVEN

VISION

Kubernetes has become a standard de facto for the enterprise infrastructure management, especially for microservice-based infrastructures.

Kubernetes operators have become a common way to extend Kubernetes with domain objects and domain logic.

At the moment (2018-2019), operators are mostly written in Go and require advanced knowledge both of Go and Kubernetes internals. This raises the entry barrier to the operator development field.

In a perfect world of Kopf, Kubernetes operators are a commodity, used to build the domain logic on top of Kubernetes fast and with ease, requiring little or no skills in infrastructure management.

For this, Kopf hides the low-level infrastructure details from the user (i.e. the operator developer), exposing only the APIs and DSLs needed to express the user's domain.

Besides, Kopf does this in one of the widely used, easy to learn programming languages: Python.

But Kopf does not go too far in abstracting the Kubernetes internals away: it avoids the introduction of extra entities and controlling structures (Occam's Razor, KISS), and most likely it will never have a mapping of Python classes to Kubernetes resources (like in the ORMs for the relational databases).

CHAPTER

FORTYEIGHT

NAMING

Kopf is an abbreviation either for **K**ubernetes **O**perator **P**ythonic **F**ramework, or for **K**ubernetes **OP**erator **F**ramework — whatever you like more.

"Kopf" also means "head" in German.

It is capitalised in natural language texts:

> *I like using Kopf to manage my domain in Kubernetes.*

It is lower-cased in all system and code references:

```
pip install kopf
import kopf
```

CHAPTER

FORTYNINE

ALTERNATIVES

49.1 Metacontroller

The closest equivalent of Kopf is Metacontroller. It targets the same goal as Kopf does: to make the development of Kubernetes operators easy, with no need for in-depth knowledge of Kubernetes or Go.

However, it does that in a different way than Kopf does: with a few YAML files describing the structure of your operator (besides the custom resource definition), and by wrapping your core domain logic into the Function-as-a-Service or into the in-cluster HTTP API deployments, which in turn react to the changes in the custom resources.

An operator developer still has to implement the infrastructure of the API calls in these HTTP APIs and/or Lambdas. The APIs must be reachable from inside the cluster, which means that they must be deployed there.

Kopf, on the other hand, attempts to keep things explicit (as per the Zen of Python: *explicit is better than implicit*), keeping the whole operator's logic in one place, in one syntax (Python).

And, by the way…

Not only it is about "*explicit is better than implicit*", but also "*simple is better than complex*", "*flat is better than nested*", and "*readability counts*", which makes Kopf a *pythonic* framework in the first place, not just *written with Python*.

Kopf also makes the effort to keep the operator development human-friendly, which means at least the ease of debugging (e.g. with the breakpoints, running in a local IDE, not in the cloud), the readability of the logs, and other little pleasant things.

And also Kopf allows to write *any* arbitrary domain logic of the resources, especially if it spans over long periods (hours, days if needed), and is not limited to the timeout restrictions of the HTTP APIs with their expectation of nearly-immediate outcome (i.e. in seconds or milliseconds).

Metacontroller, however, is more mature, 1.5 years older than Kopf, and is backed by Google, who originally developed Kubernetes itself.

Unlike Kopf, Metacontroller supports the domain logic in any languages due to its language-agnostic nature of HTTP APIs.

49.2 Side8's k8s-operator

Side8's k8s-operator is another direct equivalent. It was the initial inspiration for writing Kopf.

Side8's k8s-operator is written with Python3 and allows to write the domain logic in the apply/delete scripts in any language. The scripts run locally on the same machine where the controller is running (usually the same pod, or a developer's computer).

However, the interaction with the script relies on stdout output and the environment variables as the input, which is only good if the scripts are written in shell/bash. Writing the complicated domain logic in bash can be troublesome.

The scripts in other languages, such as Python, are supported but require the inner infrastructure logic to parse the input and to render the output and to perform the logging properly: e.g., so that no single byte of garbage output is ever printed to stdout, or so that the resulting status is merged with the initial status, etc – which kills the idea of pure domain logic and no infrastructure logic in the operator codebase.

49.3 CoreOS Operator SDK & Framework

CoreOS Operator SDK is not an operator framework. It is an SDK, i.e. a Software Development Kit, which generates the skeleton code for the operators-to-be, and users should enrich it with the domain logic code as needed.

CoreOS Operator Framework, of which the abovementioned SDK is a part, is a bigger, more powerful, but very complicated tool for writing operators.

Both are developed purely for Go-based operators. No other languages are supported.

From the CoreOS'es point of view, an operator is a method of packaging and managing a Kubernetes-native application (presumably of any purpose, such as MySQL, Postgres, Redis, ElasticSearch, etc) with Kubernetes APIs (e.g. the custom resources of ConfigMaps) and `kubectl` tooling. They refer to operators as "*the runtime that manages this type of application on Kubernetes*."

Kopf uses a more generic approach, where the operator *is* the application with the domain logic in it. Managing other applications inside of Kubernetes is just one special case of such a domain logic, but the operators could also be used to manage the applications outside of Kubernetes (via their APIs), or to implement the direct actions without any supplementary applications at all.

See also:

- https://coreos.com/operators
- https://coreos.com/blog/introducing-operator-framework
- https://enterprisersproject.com/article/2019/2/kubernetes-operators-plain-english

CHAPTER

FIFTY

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W