Kopf

Sergey Vasilyev

Jun 03, 2026

FIRST STEPS:

CHAPTER

ONE

INSTALLATION

Prerequisites:

- Python >= 3.10 (CPython and PyPy are officially tested and supported).
- A Kubernetes cluster (k3d/k3s, minikube, OrbStack, Docker, AWS, GCP, etc).

1.1 Running without installation

1.1.1 Using uvx

To run Kopf as a tool via uvx with an operator:

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

1.1.2 Using docker

A pre-built Docker image with all extras is available for quick experimentation — no local Python installation needed:

```
# Minimize the credentials exposure.
kubectl config view --minify --flatten > dev.kubeconfig

# Run the operator locally, target a local cluster (host networking).
docker run --rm -it --network=host \
    -v ./examples/01-minimal/example.py:/app/main.py:ro \
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf
```

See *Docker image* for image variants, tags, and more usage examples.

1.2 Installing packages

1.2.1 Using pip

To install Kopf using `pip` (assuming you have a virtualenv activated):

```
pip install kopf
```

1.2.2 Using uv

If you use uv, once you are ready to build a project, add Kopf as a dependency:

```
uv add kopf
```

1.3 Cluster preparation

Unless you use the standalone mode, create a few Kopf-specific custom resources in the cluster. These are used to coordinate several instances of Kopf-based operators so that they do not double-process the same resources — only one operator will be active at a time:

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

Depending on the security rules of your cluster, you might need *RBAC* resources applied, too (cluster- and user-specific; not covered here).

1.4 Example operators

To run the example operators or the code snippets from the documentation, apply this CRD for `kopf.dev/v1/kopfexamples`. The examples will not operate without it, but your own operator with your own resources does not need it:

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

You are ready to go. Run an operator using `pip` and `virtualenv`:

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

Using `uv`:

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

1.5 Extras

To minimize the disk footprint of Kopf projects, some heavy dependencies are omitted by default. You can add them as extras if you need them.

1.5.1 `full-auth`

If you use a managed Kubernetes service that requires sophisticated authentication beyond plain username+password, fixed tokens, or client SSL certificates, add the `full-auth` extra with Kubernetes clients, which include those sophisticated authentication methods (also see *authentication piggy-backing*):

```
pip install 'kopf[full-auth]'
uv add 'kopf[full-auth]'
uvx --from 'kopf[full-auth]' kopf run -v examples/01-minimal/example.py
```

1.5.2 uvloop

If you want extra I/O performance, install the uvloop extra (also see *Custom event loops*). It will be activated automatically when installed; no extra flags or configuration are needed:

```
pip install 'kopf[uvloop]'
uv add 'kopf[uvloop]'
uvx --from 'kopf[uvloop]' kopf run -v examples/01-minimal/example.py
```

1.5.3 dev

If you want mutating/validating webhooks (also see *Admission control*) with self-signed certificates and/or ngrok tunneling, install with the `dev` extra:

```
pip install 'kopf[dev]'
uv add 'kopf[dev]'
uvx --from 'kopf[dev]' kopf run -v examples/01-minimal/example.py
```

Warning

Self-signed certificates are unsafe for production environments. Ngrok tunnelling is not needed in production environments. This extra is intended for development environments only. Hence the name.

Note

This is the `dev` extra, not the `dev` dependency group. The dependency groups are available only when installing Kopf from source.

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 arbitrary custom resources and custom controllers.

At the top level, it consists of the Kubernetes API, through which 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 behavior) 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 controller, which orchestrates objects of a specific kind, with some domain logic implemented inside.

The essential difference between operators and controllers is that operators are domain-specific controllers, but not all controllers are necessarily operators: for example, the built-in controllers for pods, deployments, services, etc, as well 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 exactly what it does.

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 solve a small real-world problem with Kopf step by step, presenting and explaining Kopf features one by one.

3.1 Problem Statement

In Kubernetes, there are no ephemeral volumes of large sizes, e.g. 500 GB. By ephemeral, we mean that the volume does not persist after it is used. Such volumes can serve 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 — meaning it is not deleted after use and can only be removed manually.

There is StatefulSet, which has a volume claim template, but the volume claim is again persistent, and StatefulSets follow the same flow as Deployments, not Jobs.

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 one or more 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 cases when the claim was never used — e.g. if the pod could not start for some reason — 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.
  - For at least N seconds to allow the pods to start safely.
  - For at most M seconds in case the pod failed to start but the claim was already 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, install the following tools locally (e.g. on macOS):

- Python >= 3.10 (running in a venv is recommended, though it 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: you can use them as 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

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

Let us 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
```

Note that the short names are those specified in 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 object. Let us 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
from typing import Any

@kopf.on.create('ephemeralvolumeclaims')
def create_fn(body: kopf.Body, **_: Any) -> None:
    logging.info(f"A handler is called with body: {body}")
```

Note

Despite an obvious desire, do not name the file `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 us run the operator and see what happens:

```
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.
```

(continues on next page)

(continued from previous page)

```
[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␣
↪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 detected an object that was created before the operator itself was started, and handled it because it had not been 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 again, as it has already been successfully handled. This is important when the operator is restarted — whether running as a deployed pod or restarted manually for debugging.

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

Listing 3: ephemeral.py

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

(continues on next page)

(continued from previous page)

```
import yaml
from typing import Any

@kopf.on.create('ephemeralvolumeclaims')
def create_fn(spec: kopf.Spec, name: str, namespace: str | None, logger: kopf.Logger, **_
↪: Any) -> None:

    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}")
```

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 the EVC.

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 `PersistentVolumeClaim` (PVC).

What happens if we change the size of the EVC after 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 us 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: kopf.Spec, name: str, namespace: str | None, logger: kopf.Logger, **_
↪: Any) -> dict[str, str]:

    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 does not 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 information on field pruning, see the Kubernetes docs.

Let us add 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:

```
import kopf
from typing import Any

@kopf.on.update('ephemeralvolumeclaims')
def update_fn(spec: kopf.Spec, status: kopf.Status, namespace: str | None, logger: kopf.
↪Logger, **_: Any) -> None:

    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 us 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` incorrectly show the capacity of PVCs: it remains the same (1G) even after the change. What matters is the size of the actual PV (Persistent Volume) behind each PVC. This is a Kubernetes issue unrelated to Kopf, so we work around it.

CHAPTER

NINE

DIFFING THE FIELDS

Previously (*Updating the objects*), we set up cascaded updates so that the PVC size is updated every time the EVC size changes.

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 us re-label the PVC with the labels of its EVC, and keep them in sync. The sync is one-way: relabeling 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: Any, new: Any, status: kopf.Status, namespace: str | None, **_: Any) ->␣
↪None:

    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, you need to set it to `None` in the patch object.

So, we need to know which fields were deleted from the EVC. Kubernetes does not natively provide this information in object events, since it notifies operators only with the latest state of the object — as seen in the *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 us use this feature to explicitly react to the relabeling of the EVCs. Note that the `new` value for a removed dict key is `None`, which is exactly what the patch object needs to delete that field:

Listing 2: ephemeral.py

```
@kopf.on.field('ephemeralvolumeclaims', field='metadata.labels')
def relabel(diff: kopf.Diff, status: kopf.Status, namespace: str | None, **_: Any) ->␣
↪None:

    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 unrelated labels placed on the PVC — e.g. manually, from a template, or by other controllers/operators, besides the labels coming from the parent EVC — are preserved and never touched (unless a label with the same name is applied to the 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 take a different approach and use a built-in feature of Kubernetes: owner references.

Let us extend the creation handler:

Listing 1: ephemeral.py

```
import kopf
import kubernetes
import os
import yaml
from typing import Any

@kopf.on.create('ephemeralvolumeclaims')
def create_fn(spec: kopf.Spec, name: str, namespace: str | None, logger: kopf.Logger,␣
↪body: kopf.Body, **_: Any) -> dict[str, str]:

    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)
```

(continues on next page)

(continued from previous page)

```
    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}
```

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

The PVC is now "owned" by the EVC, meaning it has an owner reference. When the parent EVC object is deleted, the child PVC will also be automatically deleted by Kubernetes, so we do not need to manage 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 to fully clean up the cluster.

CHAPTER

TWELVE

HANDLERS

Handlers are Python functions with the actual behavior 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 does not clarify what was changed.

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

To hide the complexity of state storing, Kopf provides cause detection: whenever an event happens for the object, the framework detects what actually happened, 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
from typing import Any

@kopf.on.create('kopfexamples')
def my_handler(spec: kopf.Spec, **_: Any) -> None:
    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 are ambiguous when multiple instances exist or when multiple sub-classes inherit from the class, thus inheriting the handlers).

If 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 ambiguity and giving meaning to `self`/`cls`:

```
import kopf
from typing import Any

class MyCls:
    def my_handler(self, spec: kopf.Spec, **_: Any) -> None:
        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
from typing import Any

@kopf.on.event('kopfexamples')
def my_handler(event: kopf.RawEvent, **_: Any) -> None:
    pass
```

The event has the following structure:

```
class RawBody(TypedDict, total=False):
    apiVersion: str
    kind: str
    metadata: Mapping[str, Any]
    spec: Mapping[str, Any]
    status: Mapping[str, Any]

class RawEvent(TypedDict, total=True):
    type: Literal[None, 'ADDED', 'MODIFIED', 'DELETED']
    object: RawBody
```

The event type `None` means the initial listing of the resources before the actual watch-stream begins.

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

> **Note**
>
> Kopf invokes the event handlers for *every* event received from the stream. This 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 duplicate side effects).

12.4 State-changing handlers

Kopf goes above 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**
>
> Kopf stores the status of the handlers, such as their progress, errors, or retries, in the object itself (in annotations), which triggers low-level events, but these events are not detected as separate causes, as nothing has changed *essentially*.

The following three core cause-handlers are available:

```
import kopf
from typing import Any

@kopf.on.create('kopfexamples')
def my_handler(spec: kopf.Spec, **_: Any) -> None:
    pass

@kopf.on.update('kopfexamples')
def my_handler(spec: kopf.Spec, old: Any, new: Any, diff: kopf.Diff, **_: Any) -> None:
    pass

@kopf.on.delete('kopfexamples')
def my_handler(spec: kopf.Spec, **_: Any) -> None:
    pass
```

Despite the handlers seeing the full body of the resource object, they react only to _essential_ changes, as implemented by *kopf.DiffBaseStorage* or its descendants (*Change detection*).

In particular, Kopf ignores the whole `status` stanza as non-essential, and all fields of `metadata` except for `labels` & `annotations` — the framework remains blind to changes in these fields unless explicitly told to see them. For example, to react to changes in the status of `kind: Job`:

```
import kopf
from typing import Any

@kopf.on.update('batch/v1', 'jobs', field='status')
def job_status_changes(**_: Any) -> None:
    pass
```

> **Note**
>
> Kopf's finalizers will be added to the object when delete handlers are specified. Finalizers block Kubernetes from fully deleting objects; they will only be deleted when all finalizers are removed, i.e. only if the Kopf operator is running to remove them (see *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=True` argument can be passed to the delete cause decorator.

12.5 Resuming handlers

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

```
import kopf
from typing import Any

@kopf.on.resume('kopfexamples')
def my_handler(spec: kopf.Spec, **_: Any) -> None:
    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, and 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 into 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 handlers pointing to the same function, so that this function is called either when an object is created while the operator is running, or when the operator starts while the object already exists:

```
import kopf
from typing import Any

@kopf.on.resume('kopfexamples')
@kopf.on.create('kopfexamples')
def my_handler(spec: kopf.Spec, **_: Any) -> None:
    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 cases where the resuming handlers start threads/tasks or allocate resources, and the deletion handlers stop/free them: it could happen that the resuming handlers would be executed after the deletion handlers, thus starting threads/tasks and never stopping them. For example:

```
import asyncio
import kopf
from typing import Any

TASKS: dict[str, asyncio.Task[None]] = {}

```

(continues on next page)

(continued from previous page)

```
@kopf.on.delete('kopfexamples')
async def my_handler(spec: kopf.Spec, name: str, **_: Any) -> None:
    if name and name in TASKS:
        TASKS[name].cancel()

@kopf.on.resume('kopfexamples')
@kopf.on.create('kopfexamples')
def my_handler(spec: kopf.Spec, **_: Any) -> None:
    if name and name not in TASKS:
        TASKS[name] = asyncio.create_task(some_coroutine(spec))
```

In this example, if the operator starts and notices an object that has been marked for deletion, the deletion handler will be called, but the resuming handler is not called at all, despite the object being present. 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:

```
import kopf
from typing import Any

@kopf.on.resume('kopfexamples', deleted=True)
def my_handler(spec: kopf.Spec, **_: Any) -> None:
    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
from typing import Any

@kopf.on.field('kopfexamples', field='spec.somefield')
def somefield_changed(old: Any, new: Any, **_: Any) -> None:
    pass
```

There is no special detection of the causes for the fields, such as create/update/delete, so the field handler is effective 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, as well as the handling cycle of the framework.

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

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

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

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

```
import functools
import kopf
from typing import Any

@kopf.on.create('kopfexamples')
async def create_fn(spec: kopf.Spec, **_: Any) -> None:
    fns = {}

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

   await kopf.execute(fns=fns)

def handle_item(item: Any, *, spec: kopf.Spec, **_: Any) -> None:
    pass
```

Or declaratively with decorators:

```
import kopf
from typing import Any

@kopf.on.create('kopfexamples')
def create_fn(spec: kopf.Spec, **_: Any) -> None:

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

        @kopf.subhandler(id=item)
        def handle_item(item: Any = item, **_: Any) -> None:
            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 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 stored on the object between 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 for 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.

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 kopf
import time
from typing import Any

@kopf.daemon('kopfexamples')
async def monitor_kex_async(**_: Any) -> None:
    while True:
        ...  # check something
        await asyncio.sleep(10)

@kopf.daemon('kopfexamples')
def monitor_kex_sync(stopped: kopf.DaemonStopped, **_: Any) -> None:
    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 or pausing (see *Peering*).

In both cases, Kopf requests all daemons 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
from typing import Any

@kopf.daemon('kopfexamples')
def monitor_kex(stopped: kopf.DaemonStopped, **_: Any) -> None:
    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` raised at any point of their code (specifically, at any `await` clause):

```
import asyncio
import kopf
from typing import Any

@kopf.daemon('kopfexamples', cancellation_timeout=1.0)
async def monitor_kex(**_: Any) -> None:
    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 how long the coroutine should 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
from typing import Any

@kopf.daemon('kopfexamples',
             cancellation_backoff=1.0, cancellation_timeout=3.0)
```

(continues on next page)

(continued from previous page)

```
async def monitor_kex(stopped: kopf.DaemonStopped, **_: Any) -> None:
    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
from typing import Any

@kopf.daemon('kopfexamples')
def monitor_kex(stopped: kopf.DaemonStopped, **_: Any) -> None:
    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
from typing import Any

@kopf.daemon('kopfexamples')
async def monitor_kex(stopped: kopf.DaemonStopped, **_: Any) -> None:
    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 a 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
from typing import Any

@kopf.daemon('kopfexamples', initial_delay=30)
async def monitor_kex(stopped: kopf.DaemonStopped, **_: Any) -> None:
    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.

The `initial_delay` can also be a callable, which accepts the same arguments as the handler itself, and returns the delay in seconds:

```
import kopf
import random
from typing import Any

def get_delay(body: kopf.Body, **_: Any) -> int:
    return random.randint(
        body.get('spec', {}).get('minDelay', 0),
        body.get('spec', {}).get('maxDelay', 60),
    )

@kopf.daemon('kopfexamples', initial_delay=get_delay)
async def monitor_kex(stopped: kopf.DaemonStopped, **_: Any) -> None:
    ...
```

This is primarily intended for load balancing during operator restarts (e.g. by using a random delay). If you need more complex or periodic random timing, consider using a daemon with custom sleeps instead of a timer.

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
from typing import Any

@kopf.daemon('kopfexamples')
async def monitor_kex(stopped: kopf.DaemonStopped, **_: Any) -> None:
    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 removed anyway, 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 kopf
import random
import time
from typing import Any

@kopf.daemon('kopfexamples')
def monitor_kex(stopped: kopf.DaemonStopped, logger: kopf.Logger, body: kopf.Body, spec:␣
↪kopf.Spec, **_: Any) -> None:
    while not stopped:
        logger.info(f"FIELD={spec['field']}")
        time.sleep(1)

@kopf.timer('kopfexamples', interval=2.5)
def modify_kex_sometimes(patch: kopf.Patch, **_: Any) -> None:
    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 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: int, **_: Any) -> None:
    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).

13.10 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
from typing import Any
```

(continues on next page)

(continued from previous page)

```
@kopf.daemon('kopfexamples')
async def monitor_kex(stopped: kopf.DaemonStopped, **_: Any) -> dict[str, bool]:
    await asyncio.sleep(10.0)
    return {'finished': True}
```

13.11 Patching

Daemons can modify the resource via the *patch* keyword argument, including both the merge-patch dictionary and the transformation functions (see *Patching* for details).

```
import asyncio
import kopf
import random
from typing import Any

# Transformation functions and JSON-patches are useful specifically for the lists.
def set_conditions(body: kopf.RawBody) -> None:
    conditions = body.setdefault('status', {}).setdefault('conditions', [])
    conditions[:] = [cond for cond in conditions if cond.get('type') != 'Whatever']
    conditions.append({'type': 'Whatever', 'status': 'True', 'reason': 'SomeReason',
↪'message': 'Some message'})

@kopf.daemon('kopfexamples')
async def update_status(stopped: kopf.DaemonStopped, patch: kopf.Patch, **_: Any) ->␣
↪None:
    # This goes to the merge-patch.
    patch.status['replicas'] = random.randint(1, 10)

    # This goes to the JSON-patch.
    patch.fns.append(set_conditions)

    # Exit the daemon so that it restarts again (otherwise exits forever).
    raise kopf.TemporaryError("retry a bit later", delay=5)
```

The patch is applied after the handler exits on each iteration of the run loop. This includes when the handler raises *kopf.TemporaryError* for retrying: all changes accumulated in the patch during that attempt are sent to the Kubernetes API before the next retry begins. After the patch is applied, it is cleared for the next iteration.

If a transformation function's JSON Patch hits a `resourceVersion` mismatch (HTTP 422), the transformation functions are carried forward and retried on the next iteration — not in the background. The handler can detect this by checking `bool(patch)` at the start: if it is true before the handler has made any changes, there are pending transformation functions from a previous iteration.

13.12 Filtering

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

```
import kopf
import time
from typing import Any
```

(continues on next page)

(continued from previous page)

```
@kopf.daemon('kopfexamples',
             annotations={'some-annotation': 'some-value'},
             labels={'some-label': 'some-value'},
             when=lambda name, **_: 'some' in name)
def monitor_selected_kexes(stopped: kopf.DaemonStopped, **_: Any) -> None:
    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.

13.13 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 kopf
import time
from typing import Any

@kopf.timer('kopfexamples', interval=1.0)
def ping_kex(spec: kopf.Spec, **_: Any) -> None:
    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 kopf
import time
from typing import Any

@kopf.timer('kopfexamples', interval=1.0, sharp=True)
def ping_kex(spec: kopf.Spec, **_: Any) -> None:
    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
from typing import Any

@kopf.timer('kopfexamples', idle=10)
def ping_kex(spec: kopf.Spec, **_: Any) -> None:
    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
from typing import Any

@kopf.timer('kopfexamples', idle=10, interval=1)
def ping_kex(spec: kopf.Spec, **_: Any) -> None:
    print(f"FIELD={spec['field']}")
```

14.4 Postponing

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

It is possible to postpone the invocations:

```
import asyncio
import kopf
import time
from typing import Any

@kopf.timer('kopfexamples', interval=1, initial_delay=5)
def ping_kex(spec: kopf.Spec, **_: Any) -> None:
    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.

The `initial_delay` can also be a callable, which accepts the same arguments as the handler itself, and returns the delay in seconds:

```
import kopf
import random
from typing import Any

def get_delay(body: kopf.Body, **_: Any) -> int:
    return random.randint(
        body.get('spec', {}).get('minDelay', 0),
        body.get('spec', {}).get('maxDelay', 60),
    )
```

(continues on next page)

(continued from previous page)

```
@kopf.timer('kopfexamples', interval=1, initial_delay=get_delay)
def ping_kex(spec: kopf.Spec, **_: Any) -> None:
    ...
```

This is primarily intended for load balancing during operator restarts (e.g. by using a random delay). If you need more complex or periodic random timing, consider using a daemon with custom sleeps instead of a timer.

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
from typing import Any

@kopf.timer('kopfexamples',
            initial_delay=60, interval=10, idle=600)
def ping_kex(spec: kopf.Spec, **_: Any) -> None:
    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 behavior is to retry an 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
from typing import Any

@kopf.timer('kopfexamples',
            errors=kopf.ErrorsMode.TEMPORARY, interval=10, backoff=5)
def monitor_kex_by_time(name: str, retry: int, **_: Any) -> None:
    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 kopf
import random
from typing import Any

@kopf.timer('kopfexamples', interval=10)
def ping_kex(spec: kopf.Spec, **_: Any) -> int:
    return random.randint(0, 100)
```

Note

Whenever a resulting value is serialized 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 Patching

Timers can modify the resource via the *patch* keyword argument, including both the merge-patch dictionary and the transformation functions (see *Patching* for details).

```
import asyncio
import kopf
import random
from typing import Any

# Transformation functions and JSON-patches are useful specifically for the lists.
def set_conditions(body: kopf.RawBody) -> None:
    conditions = body.setdefault('status', {}).setdefault('conditions', [])
    conditions[:] = [cond for cond in conditions if cond.get('type') != 'Whatever']
    conditions.append({'type': 'Whatever', 'status': 'True', 'reason': 'SomeReason',
↪'message': 'Some message'})

@kopf.timer('kopfexamples', interval=60)
async def update_status(patch: kopf.Patch, **_: Any) -> None:
    # This goes to the merge-patch.
    patch.status['replicas'] = random.randint(1, 10)

    # This goes to the JSON-patch.
    patch.fns.append(set_conditions)
```

The patch is applied after the handler exits on each timer iteration. This includes when the handler raises *kopf.TemporaryError* for retrying: all changes accumulated in the patch during that attempt are sent to the Kubernetes

API before the next retry begins. After the patch is applied, it is cleared for the next iteration.

If a transformation function's JSON Patch hits a `resourceVersion` mismatch (HTTP 422), the transformation functions are carried forward and retried on the next iteration — not in the background. The handler can detect this by checking `bool(patch)` at the start: if it is true before the handler has made any changes, there are pending transformation functions from a previous iteration.

14.9 Filtering

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

```
import kopf
from typing import Any

@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: kopf.Spec, **_: Any) -> None:
    pass
```

14.10 System resources

Warning

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

Although 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 forward compatibility: the framework can add new keywords in the future, and 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 related to the retrying process:

`retry` (`int`) is the sequential retry number for 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 and errors, this is the time of the first attempt.

`runtime` (`datetime.timedelta`) is the duration of the handler run — in case of retries and errors, this is measured from 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 when there are multiple decorators, possibly with different selectors and filters, for one handler function:

```
import kopf
from typing import Any

@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: Any, patch: kopf.Patch, **_: Any) -> None:
    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: Any, patch: kopf.Patch, **_: Any) -> None:
    patch_parent({'status': {param: {'done': True}}})
```

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

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

```
import kopf
from typing import Any

@kopf.on.update('KopfExample', param=10, field='spec.field')
@kopf.on.update('KopfExample', param=1, field='spec')
def fn(param: Any, **_: Any) -> None:
    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 that defines the operator's behavior. 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 own configuration.

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. Note that it is not necessarily the same as the selector used in the decorator, since 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. a namespace field for cluster-scoped objects), the fields are `None` — unlike accessing the same fields by key, which raises a `KeyError`.

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

15.5.2 Logging

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

Some 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 used internally by the framework itself, and is shared with handlers for convenience (since patching happens anyway in the framework, there is no need to make separate API calls for patching).

`patch` also provides a `fns` property for appending transformation functions that operate on the raw resource body as a mutable dictionary. This is useful for list operations and other changes that depend on the current state of the resource.

See *Patching* for details on both merge-patch and transformation functions.

15.5.4 In-memory container

`memo` is an in-memory container for arbitrary runtime-only key-value data. 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 specific resource object). For operator handlers, `memo` is shared by all handlers of the same operator and is 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 (the 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 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 the Kubernetes API; it is a dict with `['type']` and `['object']` keys.

15.7 Resource-changing kwargs

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

15.7.1 Causation

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

15.7.2 Diffing

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

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

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

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

Due to specifics of Kubernetes, `None` is interpreted as the absence of the value/field, not as a value in its own right. In diffs, this means the value did not exist before, or will not exist after the changes (for the old and 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 and 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`, handlers must not produce 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 the subresource being checked. `None` means 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 warning strings. Admission webhook handlers can populate the list with warnings, and the webhook servers/tunnels return them to Kubernetes, which shows them in `kubectl`.

See more: *Admission control*.

15.9.4 User information

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

It usually contains the keys `'username'`, `'uid'`, `'groups'`, but this may 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 any additional interpretation.

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

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

> **Note**
>
> This identifies the apiservers that send the admission request, not the user or application 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
from typing import Any

@kopf.on.create('kopfexamples')
async def create_fn(spec: kopf.Spec, **_: Any) -> None:
    await asyncio.sleep(1.0)
```

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

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

Warning

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

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

This can go unnoticed in a development environment with only a few resources and no external timeouts, but can cause serious problems in production environments under high load.

CHAPTER

SEVENTEEN

LOADING AND IMPORTING

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

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

18.1 By-name resource selectors

The following notations are supported to specify the resources to be handled. As a rule of thumb, they are designed to infer a developer's intentions as accurately as possible, in a way similar to `kubectl` semantics.

The resource name is always expected to be the rightmost positional 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 separated by a slash:

```
import kopf
from typing import Any

@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(**_: Any) -> None:
    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:

```
import kopf
from typing import Any

@kopf.on.event('kopf.dev', 'kopfexamples')
@kopf.on.event('apps', 'deployments')
def fn(**_: Any) -> None:
    pass
```

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

```
import kopf
from typing import Any

@kopf.on.event('kopfexamples.kopf.dev')
@kopf.on.event('deployments.apps')
def fn(**_: Any) -> None:
    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:

```
import kopf
from typing import Any

@kopf.on.event('v1', 'pods')
@kopf.on.event('', 'v1', 'pods')
def fn(**_: Any) -> None:
    pass
```

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

```
import kopf
from typing import Any

@kopf.on.event('kopfexamples')
@kopf.on.event('deployments')
@kopf.on.event('pods')
def fn(**_: Any) -> None:
    pass
```

In all examples above, where a resource identifier is expected, it can be any name: plural, singular, kind, or a short name. Since it is impossible to guess which one is which, the name is remembered as-is and later matched against all possible names of the specific resource once it is discovered:

```
import kopf
from typing import Any

@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(**_: Any) -> None:
    pass
```

The resource specification can be more specific on which name to match by using the keyword arguments:

```
import kopf
from typing import Any

@kopf.on.event(kind='KopfExample')
@kopf.on.event(plural='kopfexamples')
@kopf.on.event(singular='kopfexample')
@kopf.on.event(shortcut='kex')
@kopf.on.event(group='kopf.dev', plural='kopfexamples')
@kopf.on.event(group='kopf.dev', version='v1', plural='kopfexamples')
def fn(**_: Any) -> None:
    pass
```

18.2 By-category resource selectors

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

```
import kopf
from typing import Any

@kopf.on.event(category='all')
def fn(**_: Any) -> None:
    pass
```

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

18.3 Catch-all resource selectors

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

```
import kopf
from typing import Any

@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(**_: Any) -> None:
    pass
```

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

```
import kopf
from typing import Any

@kopf.on.event(kopf.EVERYTHING)
def fn(**_: Any) -> None:
    pass
```

Serving everything is better when it is used with filters:

```
import kopf
from typing import Any

@kopf.on.event(kopf.EVERYTHING, labels={'only-this': kopf.PRESENT})
def fn(**_: Any) -> None:
    pass
```

18.4 Callable resource selectors

To have fine-grained control over which resources are handled, you can use a single positional callback as the resource specifier. It must accept one positional argument of type *kopf.Resource* and return a boolean indicating whether to handle the resource:

```
import kopf
from typing import Any

def kex_selector(resource: kopf.Resource) -> bool:
    return resource.plural == 'kopfexamples' and resource.preferred

@kopf.on.event(kex_selector)
def fn(**_: Any) -> None:
    pass
```

You can combine the callable resource selectors with other keyword selectors (but not the positional by-name or catch-all selectors):

```
import kopf
from typing import Any

def kex_selector(resource: kopf.Resource) -> bool:
    return resource.plural == 'kopfexamples' and resource.preferred

@kopf.on.event(kex_selector, group='kopf.dev')
def fn(**_: Any) -> None:
    pass
```

There is a subtle difference between callable resource selectors and filters (see `when=...` in *Filtering*): a callable filter applies to all events coming from a live watch stream identified by a resource kind and a namespace (or by a resource kind alone for watch streams of cluster-wide operators); a callable resource selector decides whether to start the watch stream for that resource kind at all, which can help reduce the load on the API.

Note

Normally, Kopf selects only the "preferred" versions of each API group when filtered by names. This does not apply to callable selectors. To handle non-preferred versions, define a callable and return `True` regardless of the version or its preferred field.

18.5 Exclusion of core v1 events

Core v1 events are excluded from `EVERYTHING` and from callable selectors regardless of what the selector function returns: events are created during handling of other resources via the implicit *Events* from log messages, so they would cause unnecessary handling cycles for every meaningful change.

To handle core v1 events, name them directly and explicitly:

```
import kopf
from typing import Any

def all_core_v1(resource: kopf.Resource) -> bool:
    return resource.group == '' and resource.preferred

@kopf.on.event(all_core_v1)
@kopf.on.event('v1', 'events')
def fn(**_: Any) -> None:
    pass
```

18.6 Multiple resource selectors

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— or use a callable selector. The handlers are deduplicated by the underlying function and its handler id (which equals the function's name by default unless overridden), so a function will never be triggered multiple times for the same resource even if there are accidental overlaps in the specifications.

```
import kopf
from typing import Any

@kopf.on.event('kopfexamples')
@kopf.on.event('v1', 'pods')
def fn(**_: Any) -> None:
    pass
```

18.7 Ambiguous resource selectors

Warning

Kopf tries to make it easy to specify resources in the style of `kubectl`. However, some things cannot be made that easy. If resources are specified ambiguously — i.e. if 2 or more resources from 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 that are intended to match a specific resource by name; specifications with intentional multi-resource mode are served as usual (e.g. by categories).

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

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

While it may be convenient to write short forms of resource names, the proper approach is to always include at least an API group:

```
import kopf
from typing import Any

@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(**_: Any) -> None:
    pass
```

Reserve short forms for prototyping and experimentation, and for ad-hoc operators with custom resources (non-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 entire cluster if resources are specified without a group name (e.g. `('pods')` instead of `('v1', 'pods')`).

To avoid scanning the entire cluster and all (even unused) API groups, it is recommended to specify at least the group name 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 their 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, and each has its own 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: kopf.Spec, **_: Any) -> None:
    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: kopf.Spec, **_: Any) -> None:
    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: kopf.Spec, **_: Any) -> None:
    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(**_: Any) -> None:
    pass

@kopf.on.create('kopfexamples', field='spec.field', value=kopf.PRESENT)
def created_with_field(**_: Any) -> None:
    pass

@kopf.on.create('kopfexamples', field='spec.no-field', value=kopf.ABSENT)
def created_without_field(**_: Any) -> None:
    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(**_: Any) -> None:
    pass

@kopf.on.update('kopfexamples', field='spec.field')
def field_is_affected(old: Any, new: Any, **_: Any) -> None:
    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 and new values relevant to the specified fields, as well 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: Any, new: Any, **_: Any) -> None:
    pass
```

However, different causes —mostly resuming combined with one of creation/update/deletion— will not be distinguished, so e.g. a 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, these filtering semantics are extended for them:

The `field=` filter restricts the update handlers to cases where the specified field is affected in any way: changed, added, or removed 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 either of them satisfies the value criterion. This covers both sides of the state transition: when the value criterion has just been satisfied (but was not satisfied before),

or when the value criterion was satisfied before (but is no longer satisfied). In the latter case, 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 operator developers, except in 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 that have no concept of "updating" and being in between two equally valid and applicable states, the `field=/value=` filters check the resource in its current —and only— state. The handlers are invoked and the daemons run as long as the field and 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 and new values, which might not be what is intended. For more precise filtering, the old and new values can be checked separately with the `old=/new=` filters using the same filtering methods/markers as all other filters.

```
@kopf.on.update('kopfexamples', field='spec.field', old='x', new='y')
def field_is_edited(**_: Any) -> None:
    pass

@kopf.on.update('kopfexamples', field='spec.field', old=kopf.ABSENT, new=kopf.PRESENT)
def field_is_added(**_: Any) -> None:
    pass

@kopf.on.update('kopfexamples', field='spec.field', old=kopf.PRESENT, new=kopf.ABSENT)
def field_is_removed(**_: Any) -> None:
    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(**_: Any) -> None:
    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(**_: Any) -> None:
    pass
```

Generally, the update handlers with `old=`/`new=` filters are invoked only when the field's value changes, 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 whether the field is present or absent 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 behavior as with the unspecified `value=` filter.

Note

`value=` is currently mutually exclusive with `old=`/`new=`: only one filtering method can be used; using both together would be ambiguous. This may be reconsidered in the future.

19.4 Value callbacks

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

The value callbacks must accept the same *keyword arguments* as the respective handlers (with `**kwargs`/`**_` for forward 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: str | None, /, spec: kopf.Spec, **_: Any) -> bool:
    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: kopf.Spec, **_: Any) -> None:
    pass
```

19.5 Callback filters

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

```
def is_good_enough(spec: kopf.Spec, **_: Any) -> bool:
    return spec.get('field') in spec.get('items', [])

@kopf.on.create('kopfexamples', when=is_good_enough)
def my_handler(spec: kopf.Spec, **_: Any) -> None:
    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: kopf.Spec, **_: Any) -> None:
    pass
```

Callback filters are not limited to checking the resource's content. They can filter by any kwarg data, e.g. by the *reason* of the invocation, remembered *memo* values, etc. However, it is highly recommended that filters do not modify the state of the operator — keep that for handlers.

There is a subtle difference between callable filters and resource selectors (see *Resource specification*): a callable filter applies to all events coming from a live watch stream identified by a resource kind and a namespace (or by a resource kind alone for watch streams of cluster-wide operators); a callable resource selector decides whether to start the watch stream for that resource kind at all, which can help reduce the load on the API.

19.6 Callback helpers

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

```
import kopf
from typing import Any

def whole_fn1(name: str, **_: Any) -> bool: return name.startswith('kopf-')
def whole_fn2(spec: kopf.Spec, **_: Any) -> bool: return spec.get('field') == 'value'
def value_fn1(value: str | None, **_: Any) -> bool: return value and value.startswith(
↪'some')
def value_fn2(value: str | None, **_: Any) -> bool: return value and 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(**_: Any) -> None:
    pass

@kopf.on.create('kopfexamples',
                when=kopf.any_([whole_fn1, whole_fn2]),
                labels={'somelabel': kopf.any_([value_fn1, value_fn2])})
def create_fn2(**_: Any) -> None:
    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, no messages will be logged and no annotations will be stored on the object. Such objects are processed in stealth mode even if the operator technically sees them in the watch stream.

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

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

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

CHAPTER

TWENTY

PATCHING

Handlers can modify the Kubernetes resource they are handling by using the *patch* keyword argument. There are two patching strategies available: the merge-patch dictionary for simple field changes, and transformation functions for operations that depend on the current state of the resource, such as list manipulations.

The changes from both strategies are mixed with the framework's own modifications (such as progress storage, finalizer management, and handler result delivery) and applied together in the minimal number of API calls (depending on the resource definition).

There can be anywhere from zero to four patches per processing cycle, depending on whether the status is a subresource, and whether the patch is a mix of dictionary changes and transformation functions with actual changes.

Alternatively, operator developers can use any third-party Kubernetes client library to patch their resources directly inside the handlers instead of using the provided *patch* facility.

20.1 Dictionary merge-patches

The *patch* object behaves as a mutable dictionary. The changes accumulated in it are applied to the resource as a JSON merge-patch (`application/merge-patch+json`) after the handler finishes:

```
import kopf
from typing import Any

@kopf.on.create('kopfexamples')
def ensure_defaults(spec: kopf.Spec, patch: kopf.Patch, **_: Any) -> None:
    if 'greeting' not in spec:
        patch.spec['greeting'] = 'hello'
    patch.status['state'] = 'initialized'
```

Setting a field to `None` deletes it from the resource. Nested dictionaries are merged recursively. Other values overwrite the existing ones:

```
import kopf
from typing import Any

@kopf.on.update('kopfexamples')
def cleanup_obsolete_fields(patch: kopf.Patch, **_: Any) -> None:
    patch.spec['obsoleteField'] = None  # deletes the field
    patch.status['phase'] = 'updated'   # overwrites the value
```

20.2 Transformation functions

Some changes cannot be expressed as a merge patch. In particular, list operations (appending to or removing from a list) require knowing the current state of the list to calculate the correct indices. For example, adding a finalizer requires knowing how many finalizers are already present; removing one requires knowing its position in the list.

For these cases, the *patch* object provides the `patch.fns` property. You can append any function of type `Callable[[dict], None]` to `patch.fns` (or insert into the beginning or the middle of the list, if that matters). Each function accepts the raw resource body as a positional argument (a regular mutable dictionary) and mutates it in place. The dictionary being mutated is already a deep copy of the original body, so there is no need to worry:

```
import kopf
from typing import Any

def add_finalizer(body: kopf.RawBody, /) -> None:
    finalizers = body.setdefault('metadata', {}).setdefault('finalizers', [])
    if 'my-operator/cleanup' not in finalizers:
        finalizers.append('my-operator/cleanup')

@kopf.on.create('kopfexamples')
def create_fn(patch: kopf.Patch, **_: Any) -> None:
    patch.fns.append(add_finalizer)
```

The framework calls the transformation functions against the freshest seen resource body and computes a JSON diff (`application/json-patch+json`) relative to that body. The resulting JSON Patch operations are sent to the Kubernetes API with an optimistic concurrency check on the `metadata.resourceVersion`.

> **Note**
>
> The body passed to the transformation function is the latest version of the resource known to the framework at the time the function is applied. It may already reflect the results of earlier patch operations in the same or previous processing cycles, so it is not necessarily the body from the event that triggered the handler.

The transformation functions may be called more than once across the same or several processing cycles — for instance, if the API server rejects the patch due to a conflict. The functions should therefore be safe to call repeatedly: they should check the current state before making changes rather than assuming a particular initial state.

The transformation function takes a single positional argument for the body. If additional positional or keyword arguments are needed, use `functools.partial`:

```
import functools
import kopf
from typing import Any

def set_label(body: kopf.RawBody, /, name: str, value: str) -> None:
    body.setdefault('metadata', {}).setdefault('labels', {})[name] = value

@kopf.on.create('kopfexamples')
def create_fn(patch: kopf.Patch, **_: Any) -> None:
    patch.fns.append(functools.partial(set_label, name='my-label', value='my-value'))
```

20.3 Patch timing in daemons and timers

For *daemons* and *timers*, the patch is applied after the handler exits on each iteration of the run loop — including when the handler raises `kopf.TemporaryError` for retrying. After the patch is applied, it is cleared for the next iteration.

This means any changes accumulated in the patch dictionary and any transformation functions appended to `patch.fns` during the handler's execution are sent to the Kubernetes API before the next invocation of the handler starts.

If a transformation function's JSON Patch is rejected by the API server due to an optimistic concurrency conflict (HTTP 422), the transformation functions are carried forward to the next iteration, where they are retried against the newer state of the resource. The retry does not happen in the background — it waits until the handler is invoked again on the next timer interval or daemon retry. Handlers can detect carried-forward transformation functions by checking `bool(patch)` at the start of the handler: if it is true before the handler has made any changes, it means there are pending transformation functions from a previous iteration.

CHAPTER

TWENTYONE

RESULTS DELIVERY

All handlers can return arbitrary JSON-serializable values. Kopf then stores these values in the resource status under the name of the handler:

```
import kopf
from typing import Any

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

@kopf.on.create('kopfexamples')
def create_kex_2(uid: str, **_: Any) -> dict[str, int]:
    return {'r1': random.randint(0, 100), 'r2': random.randint(100, 999)}
```

These results are visible 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 the resource itself, given that handlers do not know in which order they will be invoked (due to error handling and retrying), and to enable recovery in case of operator failures and restarts:

```
import kopf
import pykube
from typing import Any

@kopf.on.create('kopfexamples')
def create_job(status: kopf.Status, **_: Any) -> dict[str, str]:
    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())
```

(continues on next page)

(continued from previous page)

```
    obj = pykube.Job(api, {...})  # use pvc_name here
    obj.create()
    return {'name': obj.name}

@kopf.on.create('kopfexamples')
def create_pvc(**_: Any) -> dict[str, str]:
    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 second) would make it work smoothly in most cases, until PVC creation fails for any temporary reason and has to be retried. The whole thing will eventually succeed in 1-2 additional retries, just with less friendly messages and stack traces.

CHAPTER

TWENTYTWO

ERROR HANDLING

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

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.

22.1 Temporary errors

If a raised exception 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
from typing import Any

@kopf.on.create('kopfexamples')
def create_fn(spec: kopf.Spec, **_: Any) -> None:
    if not is_data_ready():
        raise kopf.TemporaryError("The data is not yet ready.", delay=30)
```

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).

The default delay for temporary errors is hard-coded to 60 seconds and cannot be configured globally for the operator (unlike `settings.execution.default_backoff` for arbitrary errors). Override the default explicitly in code if needed.

> **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.

22.2 Permanent errors

If a raised exception inherits from *kopf.PermanentError*, the handler is considered non-retriable, non-recoverable, and permanently 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
from typing import Any

@kopf.on.create('kopfexamples')
def create_fn(spec: kopf.Spec, **_: Any) -> None:
    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.

22.3 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 the default behavior, 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
from typing import Any

@kopf.on.create('kopfexamples', errors=kopf.ErrorsMode.PERMANENT)
def create_fn(spec: kopf.Spec, **_: Any) -> None:
    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).

22.4 Timeouts

The overall runtime of the handler can be limited:

```
import kopf
from typing import Any

@kopf.on.create('kopfexamples', timeout=60*60)
def create_fn(spec: kopf.Spec, **_: Any) -> None:
    raise kopf.TemporaryError(delay=60)
```

If the handler has not succeeded within this time, it is considered to have 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.

22.5 Retries

The number of retries can be limited too:

```
import kopf
from typing import Any

@kopf.on.create('kopfexamples', retries=3)
def create_fn(spec: kopf.Spec, **_: Any) -> None:
    raise Exception()
```

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

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

22.6 Backoff

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

```
import kopf
from typing import Any

@kopf.on.create('kopfexamples', backoff=30)
def create_fn(spec: kopf.Spec, **_: Any) -> None:
    raise Exception()
```

The default is 60 seconds, which you can configure globally for the operator with `settings.execution.default_backoff`:

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.execution.default_backoff = 30
```

Note

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

CHAPTER

TWENTYTHREE

SCOPES

23.1 Namespaces

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

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

The operator can serve multiple namespaces:

```
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: the operator serves all namespaces 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 specific namespaces do (meaning all namespaces with the `,*?!` characters are excluded).

If a namespace does not exist, Kubernetes permits watching 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 a nonexistent namespace) and the resources will be served once the namespace is created.

23.2 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

TWENTYFOUR

IN-MEMORY CONTAINERS

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

24.1 Resource memos

Every resource handler gets a *memo* kwarg of type *kopf.Memo*. It is an in-memory container for arbitrary runtime-only key-value pairs. 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
from typing import Any

@kopf.on.event('KopfExample')
def pinged(memo: kopf.Memo, **_: Any) -> None:
    memo.counter = memo.get('counter', 0) + 1

@kopf.timer('KopfExample', interval=10)
def tick(memo: kopf.Memo, logger: kopf.Logger, **_: Any) -> None:
    logger.info(f"{memo.counter} events have been received in 10 seconds.")
    memo.counter = 0
```

24.2 Operator memos

In operator handlers, such as 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 populated in the startup handlers, passed from outside the operator when *Embedding* is used, or both:

```
import kopf
import queue
import threading
from typing import Any

@kopf.on.startup()
def start_background_worker(memo: kopf.Memo, **_: Any) -> None:
    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, **_: Any) -> None:
    memo['my_queue'].put(None)
    memo['my_thread'].join()

def background(queue: queue.Queue) -> None:
    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 per-resource memos. All keys and values are shallow-copied into each resource's memo, where they can be mixed with per-resource values:

```
# ... continued from the previous example.
@kopf.on.event('KopfExample')
def pinged(memo: kopf.Memo, namespace: str | None, name: str, **_: Any) -> None:
    if not memo.get('is_seen'):
        memo.my_queue.put(f"{namespace}/{name}")
        memo.is_seen = True
```

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

However, due to shallow copying, 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 visible in all individual resource handlers and daemons that use their per-resource containers.

24.3 Custom memo classes

For embedded operators (*Embedding*), it is possible to use any class for memos. It is not even necessary 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 shared by all individual resources, redefine the class to return `self` when asked to make a copy, as shown below:

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

(continues on next page)

(continued from previous page)

```
from typing import Any

@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: Any) -> None:
    print(memo.create_tpl.format(**kwargs))

@kopf.on.delete('kopfexamples')
def delete_fn(memo: CustomContext, **kwargs: Any) -> None:
    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 respects, 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`.

24.4 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 two or more embedded operator tasks are running) or asyncio contextvars, except that it provides the same interface as per-resource memos.

See also

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

CHAPTER

TWENTYFIVE

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 at gathering information than individual listing requests.

25.1 Index declaration

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

```
import kopf
from typing import Any

@kopf.index('pods')
async def my_idx(**_: Any) -> Any:
    ...
```

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 direct kwargs named the same as the index (type hints are optional):

```
import kopf
from typing import Any

# ... continued from previous examples:
@kopf.timer('KopfExample', interval=5)
def tick(my_idx: kopf.Index, **_: Any) -> None:
    ...

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

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 (as are all empty collections after that — to keep the indices clean).

> **See also**
>
> *Health-checks* for probing handlers in the example above.

25.2 Index structure

An index is always a read-only *mapping* of type *kopf.Index* with arbitrary keys pointing to *collections* of type *kopf.Store*, which in turn contain arbitrary values generated by the indexing functions. The index is initially empty. Collections are never empty: they are removed when their last item 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']}
```

Indices are not nested. A 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'}]}
```

25.3 Index content

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

```
import kopf
from typing import Any

@kopf.index('pods')
async def string_keys(namespace: str | None, name: str, **_: Any) -> Any:
    return {namespace: name}
    # {'namespace1': ['pod1a', 'pod1b', ...],
    #  'namespace2': ['pod2a', 'pod2b', ...],
    #   ...]
```

Multi-value keys are possible using tuples or other hashable types:

```
import kopf
from typing import Any

@kopf.index('pods')
async def tuple_keys(namespace: str | None, name: str, **_: Any) -> Any:
    return {(namespace, name): 'hello'}
    # {('namespace1', 'pod1a'): ['hello'],
    #  ('namespace1', 'pod1b'): ['hello'],
```

(continues on next page)

(continued from previous page)

```
    #  ('namespace2': 'pod2a'): ['hello'],
    #  ('namespace2', 'pod2b'): ['hello'],
    #   ...}
```

Multiple keys can be returned at once for a single resource, and they are all merged into their respective places in the index:

```
import kopf
from typing import Any

@kopf.index('pods')
async def by_label(labels: kopf.Labels, name: str, **_: Any) -> Any:
    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, **_: Any) -> None:
    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. However, they never repeat within the same label — a label can have only one value.

25.4 Recipes

25.4.1 Unindexed collections

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

```
import kopf
from typing import Any

@kopf.index('pods')
async def pod_names(name: str, **_: Any) -> Any:
    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
from typing import Any
```

(continues on next page)

(continued from previous page)

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

25.4.2 Enumerating resources

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

One way is to collect their identities in a flat collection — useful when you mostly need to iterate over all of them without key lookups:

```
import kopf
from typing import Any

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

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

Another way is to index them by key — when index lookups happen more often than full iterations:

```
import kopf
from typing import Any

@kopf.index('pods')
async def pods_dict(namespace: str | None, name: str, **_: Any) -> Any:
    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 | None, **_: Any) ->
↪ None:
    monitored_namespace = spec.get('monitoredNamespace', namespace)
    for ns, name in pods_dict:
        if ns == monitored_namespace:
            print(f"in {ns}: {name}")
```

25.4.3 Mirroring resources

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

```
import kopf
from typing import Any

@kopf.index('deployments')
async def whole_deployments(name: str, namespace: str | None, body: kopf.Body, **_: Any)␣
↪-> Any:
    return {(namespace, name): body}

@kopf.timer('kopfexamples', interval=5)
def tick(whole_deployments: kopf.Index, **_: Any) -> None:
    deployment, *_ = whole_deployments[('kube-system', 'coredns')]
    actual = deployment.status.get('replicas')
    desired = deployment.spec.get('replicas')
    print(f"{deployment.meta.name}: {actual}/{desired}")
```

Note

Be mindful of memory consumption on large clusters and/or overly verbose objects. Pay particular attention to memory consumption from "managed fields" (see kubernetes/kubernetes#90066).

25.4.4 Indices of indices

Iterating over all keys of an index can be slow (especially when there are many keys — e.g. with thousands of pods). In such cases, an index of an index can be built: one primary index contains the real values to be used, while a secondary index contains only the keys of the primary index (in full or in part).

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

For example, suppose you want to get all container names of all pods in a namespace. In that case, the primary index indexes containers by pods' namespace and name, while the secondary index indexes pod names by namespace only:

```
import kopf
from typing import Any

@kopf.index('pods')
async def primary(namespace: str | None, name: str, spec: kopf.Spec, **_: Any) -> Any:
    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')
async def secondary(namespace: str | None, name: str, **_: Any) -> Any:
    return {namespace: name}
    # {'namespace1': ['pod1a', 'pod1b'],
    #  'namespace2': ['pod2a', 'pod2b'],
```

(continues on next page)

(continued from previous page)

```
    #   ...}

@kopf.timer('kopfexamples', interval=5)
def tick(primary: kopf.Index, secondary: kopf.Index, spec: kopf.Spec, **_: Any) -> None:
    namespace_containers: set[str] = 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 complex structures and performance requirements are rare. For simplicity and performance, nested indices are not directly provided by the framework as a built-in feature, only as this tip based on other official features.

25.5 Conditional indexing

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

If the indexing function returns `None` or does not return anything, its result is ignored and nothing is indexed. The existing values in the index are preserved as-is (this is also the case when unexpected errors occur in the indexing function with the errors mode set to `IGNORED`):

```
import kopf
from typing import Any

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

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

```
import kopf
from typing import Any

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

25.6 Errors in indexing

Indexing functions are expected to be fast and non-blocking, as they can delay the operator startup and resource processing. For this reason, when errors occur in indexing handlers, the handlers are never retried.

Arbitrary exceptions with `errors=IGNORED` (the default) cause the framework to ignore the error and keep the existing

indexed values (which are now stale). This means that new values are expected to appear soon, but the old values are good enough in the meantime (which is usually highly likely). This is the same as returning `None`, except that the exception's stack trace is also logged:

```
import kopf
from typing import Any

@kopf.index('pods', errors=kopf.ErrorsMode.IGNORED)  # the default
async def fn1(**_: Any) -> Any:
    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 future indexing by this index (so that the indexing function is not even invoked for that resource):

```
import kopf
from typing import Any

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

@kopf.index('pods')
async def fn2(**_: Any) -> Any:
    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 a specified duration (via the error's `delay` option; set to `0` or `None` for no delay). The resource is expected to be reindexed in the future, but current problems are preventing that from happening:

```
import kopf
from typing import Any

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

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

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

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

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

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

> **Warning**
>
> **There is no ideal out-of-the-box default mode for error handling:** any kind of error in the indexing functions means the index becomes inconsistent with the actual state of the cluster and its resources. Entries for matching resources are either "lost" (permanent or temporary errors) or contain possibly outdated/stale values (ignored errors) — all of these cases represent misinformation about the actual state of the cluster.
>
> The default mode is chosen to minimize index changes and reindexing in case of frequent errors — by making no changes to the index. Additionally, the stale values may still be relevant and useful to some extent.
>
> For the other two cases, operator developers must explicitly accept the risks by setting `errors=` if the operator can afford to lose the keys.

25.7 Kwargs safety

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

The trade-off is that handlers cannot use the new features until their indices are renamed to something else. Since the new features are new, existing operator code does not use them, so this is backwards compatible.

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

- System kwargs are usually one word; name your indices with two or more words.
- System kwargs are usually singular (not always); name your indices in the plural.
- System kwargs are usually nouns; using abbreviations or prefixes/suffixes (e.g. `cnames`, `rpods`) reduces the probability of collisions.

25.8 Performance

Indexing can be a CPU- and RAM-intensive operation. The data structures behind indices are designed to be as efficient as possible:

- Index lookups are O(1) — as in Python's `dict`.
- Store updates and deletions are O(1) — a `dict` is used internally.
- Overall updates and deletions are O(k), where "k" is the number of keys per object (not the total number of keys), which is fixed in most cases, so it is O(1).

Neither the number of values stored in the index nor the total number of keys affects performance (in theory).

Some performance may be lost to additional method calls on the user-facing mappings and collections that hide the internal `dict` structures. This is assumed to be negligible compared to the overall code overhead.

25.9 Guarantees

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

The indices are guaranteed to be fully populated before any other resource-related handlers are invoked in the operator. As such, even creation handlers or raw event handlers are guaranteed to have a complete indexed overview of the cluster, not just a partial snapshot from the moment they were triggered.

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

However, the indices are safe to pass to threads or tasks for later processing if those threads or tasks are started from the startup handlers mentioned above.

25.10 Limitations

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

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

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 the framework ("batteries included").

25.11 Precautions for huge clusters

Warning

On very large clusters with many resources, it is possible to hit a deadlock with indexing if the worker limit is lower than the number of resources being indexed.

All operator activities for every individual object freeze until the operator has fully indexed the cluster. This means that at startup, there will be as many workers (asyncio tasks) as there are resources matching the indexing criteria (by resource selectors and filters).

To resolve this deadlock, use one of these two solutions:

- Keep `settings.queueing.worker_limit=None` (default). Workers will still shut down after some time of inactivity (5s) and release all system resources.
- Ensure that `settings.queueing.worker_limit` is larger than the number of resources being indexed, plus some headroom.

Also, minimize the number of resources indexed with more precise filters.

Warning

Similarly, on very large clusters with many resources, there will be heavy CPU load if synchronous handlers are used for indexing.

Synchronous handlers are executed in thread pools. If too many resources rush to be indexed at operator startup, the pool can be overwhelmed. Threads are reused and indexing handlers will pass through the pool quickly, since indexing handlers are expected to be fast and purely computational. But this will not help against the initial surge.

Thread pools do not scale down automatically, so the peak thread count will remain throughout the lifetime of the operator.

To resolve the thread explosion problem, use one of these two solutions:

- Declare the handlers as `async def`, see *Async/Await*. This eliminates threading from the indexing stage entirely.
- Set `settings.execution.max_workers` to a reasonable number of allowed threads. It can be much lower than the number of resources, but this will slow down the initial indexing proportionally. There is no limit by default (`max_workers=None`).

Also, minimize the number of resources indexed with more precise filters.

See also

See *Configuration* for details on these settings.

CHAPTER

TWENTYSIX

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.

26.1 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.

26.2 Validation handlers

```
import kopf
from typing import Any

@kopf.on.validate('kopfexamples')
def say_hello(warnings: list[str], **_: Any) -> None:
    warnings.append("Verified with the operator's hook.")

@kopf.on.validate('kopfexamples')
def check_numbers(spec: kopf.Spec, **_: Any) -> None:
    if not isinstance(spec.get('numbers', []), list):
        raise kopf.AdmissionError("Numbers must be a list if present.")
```

(continues on next page)

(continued from previous page)

```
@kopf.on.validate('kopfexamples')
def convertible_numbers(spec: kopf.Spec, warnings: list[str], **_: Any) -> None:
    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: kopf.Spec, **_: Any) -> None:
    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.

26.3 Mutation handlers

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

```
import kopf
from typing import Any

@kopf.on.mutate('kopfexamples')
def ensure_default_numbers(spec: kopf.Spec, patch: kopf.Patch, **_: Any) -> None:
    if 'numbers' not in spec:
        patch.spec['numbers'] = [1, 2, 3]

@kopf.on.mutate('kopfexamples')
def convert_numbers_if_possible(spec: kopf.Spec, patch: kopf.Patch, **_: Any) -> None:
    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: Any) -> Any:
    try:
        return float(v)
    except ValueError:
        return v
```

The semantics are the same as, or as close as possible to, the Kubernetes API's. `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.

26.4 Handler options

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

`persistent` (`bool`) — persistent webhooks will not be removed from the managed configurations on exit; non-persistent webhooks will be removed if possible. Such webhooks will block 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-persistent webhooks might be persisted and block the admissions.

`operation` (`str`) — configures 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`) — reacts only 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 is 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 from the handler itself (rejected admissions), as well 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 stabilized.

26.5 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 admission requests on resource creation, a memo is created and discarded immediately. This means that creation memos are currently useless.

This may change in the future: the memos of resource creation attempts will be preserved for a limited but short time (configurable), so that values can be shared between the admission and the handling, without causing memory leaks if the resource never succeeds in admission.

26.6 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
from typing import Any

@kopf.on.validate('kopfexamples')
def ensure_default_numbers(spec: kopf.Spec, warnings: list[str], **_: Any) -> None:
    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's StdLib `warnings` module 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 module is documented as thread-unsafe (and therefore task-unsafe) and requires hacking 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.

26.7 Admission errors

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

One and only one error is returned to the user making an API request. 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: admission errors first, then permanent errors, then temporary errors, then arbitrary errors — to select the single error to report in the admission review response.

```
@kopf.on.validate('kopfexamples')
def validate1(spec: kopf.Spec, **_: Any) -> None:
    if spec.get('field') == 'value':
        raise kopf.AdmissionError("Meh! I do not 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 do not like it. Change the field.
```

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

26.8 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, **_: Any) -> None:
    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.

26.9 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). This 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. Kopf cannot make it "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 the numerous ways in which 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, **_: Any) -> None:
    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`.
- *kopf.WebhookDockerDesktopServer* for the DockerDesktop cluster, accessing the server via a magical hostname `host.docker.internal`.

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

- *kopf.WebhookNgrokTunnel* establishes 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]`.

26.10 Authenticate apiservers

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

Furthermore, 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 authentication:

- *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 certificate 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) []:
```

(continues on next page)

(continued from previous page)

```
# 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
from typing import Any

@kopf.on.startup()
def config(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.admission.managed = 'auto.kopf.dev'
    settings.admission.server = kopf.WebhookServer(cafile='client-cert.pem')

@kopf.on.validate('kex')
def show_auth(headers: kopf.Headers, sslpeer: kopf.SSLPeer, **_: Any) -> None:
    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
```

(continues on next page)

(continued from previous page)

```
# {"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.

26.11 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, **_: Any) -> None:
    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 and use it as a CA:

```
@kopf.on.startup()
def config(settings: kopf.OperatorSettings, **_: Any) -> None:
    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, **_: Any) -> None:
    settings.admission.server = kopf.WebhookServer(
        cafile='/secrets/ca.pem',       # or cadata, or capath.
        certfile='/secrets/cert.pem',
        pkeyfile='/secrets/pkey.pem',
        password='...',                 # for the private key, if used.
        file_check_interval=60,
    )
```

You can use cert-manager or other externally provided certificate files at their known (mounted) locations without the full restart of the operator. Once any of the specified files changes, e.g. due to certificate or private key automated renewal, the webhook server will restart with the new certificate (at the latest after `file_check_interval` seconds, which defaults to 60s).

> **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 that require paid subscriptions):

```
@kopf.on.startup()
def config(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.admission.server = kopf.WebhookServer(insecure=True)
```

26.12 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 function:

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

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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, **_: Any) -> None:
    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 happens inside the callback or how it works. The exact implementation may change in the future without warning: for example, the framework may either invoke the admission handlers directly in the callback or queue the request for 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`, and 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 serves requests at the same endpoint. In rare cases where 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 retain control by running the server or by sleeping. To sleep forever, use `await asyncio.Event().wait()`. If the server/tunnel exits unexpectedly, the whole operator will exit as well.

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
```

26.13 System resource cleanup

It is recommended that custom servers/tunnels clean up the system resources they allocate at runtime. The easiest way is the `try-finally` block — the cleanup will happen on garbage collection of the generator object (beware: this 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) -> None:
        super().__init__()
        self._resource = None

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

    async def __aexit__(self, exc_type: type | None, exc_val: BaseException | None, exc_
↪tb: object) -> bool:
        self._resource.cleanup()
        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 that will actually be used. That way, the context manager acts as 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. Making webhook servers/tunnels reentrant has no practical benefit.

Note

An implementation note: webhook servers and tunnels provided by Kopf use a small 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' own webhook servers/tunnels do not need this level of complication.

See also

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

CHAPTER

TWENTYSEVEN

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
from typing import Any

LOCK: asyncio.Lock

@kopf.on.startup()
async def startup_fn(logger: kopf.Logger, **_: Any) -> None:
    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

TWENTYEIGHT

SHUTDOWN

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

```
import kopf
from typing import Any

@kopf.on.cleanup()
async def cleanup_fn(logger: kopf.Logger, **_: Any) -> None:
    pass
```

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

Similarly, the cleanup handlers are not executed if the operator is force-killed with no opportunity 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 finish completely, 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

TWENTYNINE

HEALTH-CHECKS

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

29.1 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.

29.2 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 difference between readiness and liveness probing. For operators, readiness probing makes no practical sense, as operators do not serve traffic through load balancing or services. Liveness probing can help

in catastrophic cases (e.g. the operator is stuck), but will not help with partial failures (e.g. one API call is stuck). You can read more here: https://srcco.de/posts/kubernetes-liveness-probes-are-dangerous.html

Warning

Make sure that exactly one pod of an operator is running at a time, especially during restarts — see *Deployment*.

29.3 Probe handlers

The content of the response is empty by default. It can be populated with probing handlers:

```
import datetime
import kopf
import random
from typing import Any

@kopf.on.probe(id='now')
def get_current_timestamp(**_: Any) -> str:
    return datetime.datetime.now(datetime.timezone.utc).isoformat()

@kopf.on.probe(id='random')
def get_random_value(**_: Any) -> int:
    return random.randint(0, 1_000_000)
```

The probe handlers will be executed on requests to the liveness URL, and the results will be cached for a reasonable time to prevent overloading from 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 is running at all. It can happen that some of the operator's tasks, threads, or streams break, freeze, or become unresponsive while the health-reporting task continues to run. The probability of this is low, but not zero.

There are no checks that the operator is actually processing anything (unless explicitly implemented with probe handlers), as there are no reliable criteria for that — a total absence of handled resources or events can be an expected state of the cluster.

CHAPTER

THIRTY

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 may not be enough in some setups and environments. Kopf does not attempt 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.

30.1 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
from typing import Any

@kopf.on.login()
def login_fn(**_: Any) -> kopf.ConnectionInfo | None:
    return kopf.ConnectionInfo(
        server='https://localhost',
        proxy_url='http://localhost:8080',
        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'...',
```

(continues on next page)

(continued from previous page)

```
        private_key_data=b'...',
        trust_env=True,
        expiration=datetime.datetime(2099, 12, 31, 23, 59, 59),
    )
```

Both TZ-naive and 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
from typing import Any

@kopf.on.login()
async def login_fn(**_: Any) -> kopf.ConnectionInfo | None:
    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 cases where this is implied.
- HTTP/HTTPS proxy url, possibly with credentials.

Note

Proxy support is limited to what `aiohttp` supports. Specifically, it supports plain HTTP proxies, some limited HTTPS proxies, but not SOCKS5. If you need more sophisticated proxying or tunneling, implement it as a custom HTTP session with a custom connector, for example using aiohttp_socks (Kopf claims no responsibility for the quality of this library; do your own due diligence).

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 theoretically possible to mix all authentication methods at once or to have none at all. If the credentials are inconsistent or invalid, permanent re-authentication will occur.

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.

The connection info does **not** respect environment variables by default, such as `HTTP_PROXY`, `HTTPS_PROXY`, `NO_PROXY`, or `~/.netrc`. The easiest way to enable this is to set `settings.networking.trust_env = True` in a startup handler (see *Configuration*). The built-in login handlers will propagate this setting to *kopf.ConnectionInfo* automatically.

Custom login handlers can set `trust_env=True` directly on the returned *kopf.ConnectionInfo* (see example above). As a last resort, advanced users can provide a custom `aiohttp` session with `trust_env=True` (see examples below).

30.2 Custom HTTP sessions

Advanced users can provide their own `aiohttp` client session instead of the pre-built one by returning an instance of *kopf.AiohttpSession* from the login handlers.

You can, for example, inject extra headers, remove or replace the existing ones, add sophisticated authentication schemes, or set the networking parameters, such as timeouts or connection limits (for client-side rate-limiting).

However, if you provide a custom session, you must configure the authentication yourself. This includes the username/password, tokens, or SSL certificates. Kopf will not modify the provided session (except for injecting `User-Agent`), and cannot do so: most of these fields are either hidden by `aiohttp`, or are read-only, so they can only be set at the session creation.

For your convenience, *kopf.ConnectionInfo* from the existing login functions —see *Piggybacking*— provides the methods to convert it to the typical components of the HTTP sessions:

- *kopf.ConnectionInfo.as_aiohttp_basic_auth()* for username/password.
- *kopf.ConnectionInfo.as_http_headers()* for all tokens.
- *kopf.ConnectionInfo.as_ssl_context()* for CA & SSL client certificates.

> **Note**
>
> *kopf.ConnectionInfo.as_ssl_context()* will store the certificate and private key data blobs to the disk files temporarily for a brief time, since Python's `ssl` can only load it from files, not from data blobs. It will delete the files as soon as the SSL context is constructed.

You do not need to worry about the session termination or closing — Kopf will own and manage the provided session and will close it when needed.

```python
import aiohttp
import kopf
from typing import Any

@kopf.on.login()
async def login_fn(**_: Any) -> kopf.AiohttpSession:
    credentials = kopf.login_with_kubeconfig()  # or any other available method
    headers = {
        'X-Custom-Header': 'helloworld',
        'Authorization': 'VeryAdvancedAuthScheme xyz',
        'User-Agent': f'myoperator/1.2.3 kopf/{kopf.__version__}',
    }
    session = aiohttp.ClientSession(
        connector=aiohttp.TCPConnector(
            limit_per_host=10, limit=20,
            keepalive_timeout=30.
            ssl=credentials.as_ssl_context(),
        ),
        headers=credentials.as_http_headers() | headers,
        auth=credentials.as_aiohttp_basic_auth(),
```

(continues on next page)

(continued from previous page)

```
        trust_env=True,  # respect HTTP_PROXY, HTTPS_PROXY, NO_PROXY, and ~/.netrc
    )
    return kopf.AiohttpSession(
        aiohttp_session=session,
        server=credentials.server,
        default_namespace=credentials.default_namespace,
    )
```

Warning

As a rule, `aiohttp` is the internal detail of the implementation and is normally not exposed to users except for very advanced use-cases. Kopf reserves the right to change its internal HTTP library without warning or backwards compatibility. In that case, Kopf will fail if this class is returned (will raise an exception) — to prevent the unnoticed accidental damage during such upgrades. Use at your own risk.

30.3 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
from typing import Any

@kopf.on.login()
def login_fn(**kwargs: Any) -> kopf.ConnectionInfo | None:
    return kopf.login_via_pykube(**kwargs)
```

Or:

```
import kopf
from typing import Any

@kopf.on.login()
def login_fn(**kwargs: Any) -> kopf.ConnectionInfo | None:
    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 it succeeds, returns nothing, or explicitly fails):

```
import kopf
from typing import Any

@kopf.on.login(retries=3)
def login_fn(**kwargs: Any) -> kopf.ConnectionInfo | None:
    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
from typing import Any

@kopf.on.login()
def login_fn(**kwargs: Any) -> kopf.ConnectionInfo | None:
    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()*

30.4 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 re-authentication is expected to happen frequently (e.g. every few minutes), you might want to disable the logging of re-authentication (whether this is a good idea or not is for you to decide based on the specifics of your system):

```python
import kopf
import logging
from typing import Any

@kopf.on.startup()
def disable_auth_logs(**_: Any) -> None:
    logging.getLogger('kopf.activities.authentication').disabled = True
    logging.getLogger('kopf._core.engines.activities').disabled = True
```

CHAPTER

THIRTYONE

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.

31.1 Startup configuration

Every operator has its settings (even if there is more than one operator in the same process, e.g. due to *Embedding*). The settings affect how the framework behaves in detail.

The settings can be modified in the startup handlers (see *Startup*):

```
import kopf
import logging
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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*.

31.2 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.

31.3 Logging events

`settings.posting` controls which log messages are 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 kopf
import logging
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.posting.level = logging.ERROR
```

The event-posting can be disabled completely (the default is to be enabled):

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.posting.enabled = False
```

These two settings also affect *kopf.event()* and related functions: *kopf.info()*, *kopf.warn()*, *kopf.exception()* — even if they are called explicitly in the code.

By default, log messages made by the handlers on their `logger` are not posted as Kubernetes events in order to keep the events list clean. This can be enabled to improve observability if desired:

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.posting.loggers = True
```

31.4 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
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.execution.max_workers = 20
```

It is possible to replace the whole asyncio executor used for synchronous handlers (see *Async/Await*).

Note that handlers that started in a previous executor will be continued and finished with their original executor. This includes the startup handler itself. To avoid this, make the on-startup handler asynchronous:

```
import concurrent.futures
import kopf
from typing import Any

@kopf.on.startup()
async def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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
from typing import Any

@kopf.on.startup()
async def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.execution.max_workers = 1000
```

31.5 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 `settings.networking.connect_timeout` is used if set. If not set, `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 left 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, and just produces extra logs).

Warning

Some setups that involve any kind of a load balancer (LB), such as the cloud-hosted Kubernetes clusters, had a well-known problem of freezing and going silent for no reason if nothing happens in the cluster for some time. The best guess is that the connection operator<>LB remains alive, while the connection LB<>K8s closes. Kopf-based operators remain unaware of this disruption.

This was fixed in Kopf 1.44.0 by using bookmark events. Prior to 1.44.0, setting either the client or the server timeout solves the problem of recovering from such freezes, but at the cost of regular reconnections in the normal flow of operations. There is no good default value either; you should determine it experimentally based on your operational requirements, cluster size, and activity level, usually in the range of 1-10 minutes.

`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).

`settings.watching.inactivity_timeout` (seconds) is how long the watch stream is allowed to stay completely silent —delivering no events, not even bookmarks— before it is considered dead and closed for reconnection. Kubernetes sends bookmark events every 60 seconds as a heartbeat and caches the recent events for 75 seconds (here), so the default of 70 seconds allows for reasonable jitter while still detecting streams that are silently stalled at the TCP level, and then streaming the events from Kubernetes while they are still fresh. You can adjust the timeout if needed. Values lower than 62 will cause frequent reconnects on low-activity clusters. To effectively disable the inactivity tracking, set it to a huge number, e.g., `999999999` (30 years).

Note

The inactivity tracking is NOT supported in Python 3.10. It works only since Python 3.11 and higher. Python 3.10 behaves the old way — ignores the stalled connections and might freeze with no action indefinitely. As a workaround, set `settings.watching.client_timeout` to 1-10 mins. Python 3.10's end-of-life is October 2026, so the fix is not planned. However, the mere flow of bookmark events every 60 seconds may keep the connection alive and resolve the original issue of freezing.

```python
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.networking.connect_timeout = 10
    settings.networking.request_timeout = 60
    settings.watching.server_timeout = 10 * 60
```

31.5.1 Proxy and environment trust

`settings.networking.trust_env` (boolean) controls whether the HTTP client session respects the proxy-related environment variables (`HTTP_PROXY`, `HTTPS_PROXY`, `NO_PROXY`) and the `~/.netrc` file for credentials. The default is `False`.

When set to `True`, all built-in login handlers propagate this flag to *kopf.ConnectionInfo*, which in turn passes it

to the HTTP client session. The session will then use the environment variables and `~/.netrc` to configure proxies and credentials automatically. This is useful in environments where the operator must route traffic through a corporate proxy or where the credentials are managed externally.

For more details on authentication and custom login handlers, see *Authentication*.

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.networking.trust_env = True
```

31.6 Consistency

Generally, Kopf processes the resource events and updates streamed from the Kubernetes API as soon as possible, with no delays or skipping. However, high-level change-detection handlers (creation/resume/update/deletion) require a consistent state of the resource. _Consistency_ means that all patches applied by Kopf itself have arrived back via the watch-stream. If Kopf did not patch the resource recently, it is consistent by definition.

The _inconsistent_ states can happen in relatively rare circumstances on slow networks (with high latency between operator and api-servers) or under high load (high number of resources or changes), especially when an unrelated application or another operator patches the resources on their own.

Handling the _inconsistent_ states could cause double-processing (i.e. double handler execution) and some other undesired side effects. To prevent handling inconsistent states, all state-dependent handlers wait until _consistency_ is reached via one of the following two ways:

- The expected resource version from the PATCH API operation arrives via the watch-stream of the resource within the specified time window.
- The expected resource version from the PATCH API operation does not arrive via the watch-stream within the specified time window, in which case Kopf assumes consistency after the time window ends, and processing continues as if the version had arrived, possibly causing the mentioned side effects.

The time window is measured relative to the time of the latest `PATCH` call. The timeout should be long enough to assume that if the expected resource version did not arrive within the specified time, it will never arrive.

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.persistence.consistency_timeout = 10
```

The default value (5 seconds) aims for the safest scenario out of the box.

The value of `0` will effectively disable the consistency tracking and declare all resource states as consistent — even if they are not. Use this with care — e.g., with self-made persistence storages instead of Kopf's annotations (see *Handling progress* and *Change detection*).

The consistency timeout does not affect low-level handlers with no persistence, such as `@kopf.on.event`, `@kopf.index`, `@kopf.daemon`, `@kopf.timer` — these handlers run for each and every watch-event with no delay (if they match the *filters*, of course).

31.7 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
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.persistence.finalizer = 'my-operator.example.com/kopf-finalizer'
```

The default is the one that was hard-coded before: `kopf.zalando.org/KopfFinalizerMarker`.

31.8 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
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.persistence.progress_storage = kopf.StatusProgressStorage(field='status.my-
↪operator')
```

To store in multiple places (all are written to in sync, but the first found state will be used when reading, i.e. the first storage has precedence):

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.persistence.progress_storage = kopf.MultiProgressStorage([
        kopf.AnnotationsProgressStorage(prefix='my-op.example.com'),
```

(continues on next page)

(continued from previous page)

```
        kopf.StatusProgressStorage(field='status.my-operator'),
    ])
```

The default storage is in annotations, plus read-only from the status stanza, with annotations taking precedence over the status. This was a transitional solution from the original status-only storage (Mar'20–Mar'26; v0.27–1.44). The annotations are `kopf.zalando.org/{id}` (read-write), the status fields are `status.kopf.progress.{id}` (read-only). It is an equivalent of:

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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 designed to ensure a smooth upgrade of Kopf-based operators to the new state location without any special upgrade actions or conversions.

31.9 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
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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 storage unless multiple Kopf-based operators must handle the same resources and must not collide with each other. In that case, they must use different names.

31.10 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 yet handled (due to the absence of a diff-base key), or will lose 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 the 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
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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.

31.11 Cluster discovery

`settings.scanning.disabled` controls the cluster discovery at runtime.

If enabled (the default), the operator will try to observe namespaces and custom resources, and will gracefully start/stop the watch streams for them (as well as peering activities, if applicable). This requires RBAC permissions to list/watch V1 namespaces and CRDs.

If disabled, or if enabled but the permission is not granted, then only the specific namespaces will be served, with namespace patterns ignored; and only the resources detected at startup will be served, with added CRDs or CRD versions being ignored and deleted CRDs causing failures.

The default mode is sufficient for most cases, unless the strict (non-dynamic) mode is intended, for example to suppress the warnings in the logs.

If you have very restrictive cluster permissions, disable the cluster discovery:

```
@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.scanning.disabled = True
```

31.12 Retrying of API errors

In some cases, the Kubernetes API servers might not be ready on startup, or occasionally at runtime; the network might have issues too. In most cases, these issues are temporary and resolve themselves within seconds.

The framework retries TCP/SSL networking errors and HTTP 5xx errors ("the server is wrong") — i.e. everything presumed to be temporary; other errors presumed to be permanent, including HTTP 4xx errors ("the client is wrong"), escalate immediately without retrying.

The setting `settings.networking.error_backoffs` controls how many times and with what backoff interval (in seconds) the retries are performed.

It is a sequence of back-offs between attempts (in seconds):

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.networking.error_backoffs = [10, 20, 30]
```

Note that the number of attempts is one more than the number of backoff intervals (because the backoffs happen between the attempts).

A single integer or float value means a single backoff, i.e. 2 attempts total: `(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 (at 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 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
from collections.abc import Iterator
from typing import Any

class InfiniteBackoffsWithJitter:
    def __iter__(self) -> Iterator[int]:
        while True:
            yield 10 + random.randint(-5, +5)

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.networking.error_backoffs = InfiniteBackoffsWithJitter()
```

Retrying an API error blocks the task or the object's worker in which the API error occurs. However, other objects and tasks run normally in parallel (unless they encounter the same error in the same cluster).

Each consecutive error leads to the next, typically larger backoff. Each success resets the backoff intervals, and they restart from the beginning on the next error.

Note

The format is the same as for `settings.queueing.error_delays`. The only difference is that 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 indefinitely 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*.

31.13 Throttling of "too many requests"

When the API server responds with HTTP 429 "Too Many Requests", Kopf will retry as for usual errors. However, it will obey the server-suggested interval if it is longer than what Kopf would use otherwise from its own `settings.networking.error_backoffs`.

`settings.networking.enforce_retry_after` (boolean) tells Kopf what to do when its own backoff interval is longer than the server-requested interval. If `True`, the server-provided interval will be used. If `False` (default), Kopf's longer backoff interval will be used. Either way, the backoff interval will never be shorter than what the server requested.

31.14 Throttling of unexpected errors

To prevent an uncontrollable flood of activity in case of errors that prevent resources from being marked as handled, which could lead to Kubernetes API flooding, it is possible to throttle activity on a per-resource basis:

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.queueing.error_delays = [10, 20, 30]
```

In that case, all unhandled errors in the framework or in the Kubernetes API will be backed off by 10s after the 1st error, then by 20s after the 2nd, and then by 30s after the 3rd, 4th, 5th errors and so on. On the first success, the backoff intervals will be reset and reused on the next error.

The default is a sequence of Fibonacci numbers from 1 second to 10 minutes.

The backoffs are not persisted, so they are lost on operator restarts.

These backoffs do not cover errors in the handlers — the handlers have their own per-handler backoff intervals. These backoffs are for Kopf's own errors.

To disable throttling (at your own risk), set it to `[]` or `()`. This means: no throttling delays set, no throttling sleeps performed.

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.

31.15 Log levels & filters

If the logs of any component are too verbose or contain sensitive data, this can be controlled with the usual Python logging machinery.

For example, to disable the access logs of the probing server:

```
import logging
from typing import Any

@kopf.on.startup()
async def configure(**_: Any) -> None:
    logging.getLogger('aiohttp.access').propagate = False
```

To selectively filter only some log messages but not the others:

```
import kopf
import logging
from typing import Any

class ExcludeProbesFilter(logging.Filter):
    def filter(self, record: logging.LogRecord) -> bool:
        return 'GET /healthz ' not in record.getMessage()

@kopf.on.startup()
async def configure_access_logs(**_: Any) -> None:
    logging.getLogger('aiohttp.access').addFilter(ExcludeProbesFilter())
```

For more information on the logging configuration, see: logging.

In particular, you can use the special logger `kopf.objects` to filter object-related log messages coming from the *logger* and from Kopf's internals, which are then posted as Kubernetes events (`v1/events`):

```python
import kopf
import logging
from typing import Any

class ExcludeKopfInternals(logging.Filter):
    def filter(self, record: logging.LogRecord) -> bool:
        return '/kopf/' not in record.pathname

@kopf.on.startup()
async def configure_kopf_logs(**_: Any) -> None:
    logging.getLogger('kopf.objects').addFilter(ExcludeKopfInternals())
```

Warning

The path names and module names of internal modules, as well as the extra fields of `logging.LogRecord` added by Kopf, can change without notice. Do not rely on their stability. They are not a public interface of Kopf.

CHAPTER

THIRTYTWO

PEERING

All running operators communicate with each other via peering objects (an additional kind of custom resources), so they know about each other.

32.1 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 run with the same priority, all operators issue a warning and freeze, leaving the cluster unserved.

To set the operator's priority, use *--priority*:

```
kopf run --priority=100 ...
```

Or:

```
import kopf
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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.

32.2 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:

```
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.

32.3 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
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    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.

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 cluster-scoped operators handling different resource kinds, which should not be concerned with operators for other kinds.

32.4 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
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.peering.standalone = True
```

In that case, the operator will not pause if other operators with a higher priority start handling the objects, which may lead to conflicting changes and reactions from multiple operators for the same events.

32.5 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.

32.6 Multi-pod operators

Usually, one and only one operator instance should be deployed per resource type. If that operator's pod dies, handling of resources of that type will stop until the operator's pod is restarted (if it is restarted at all).

To start multiple operator pods, they must be distinctly prioritized. 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 exits, the second-highest priority operator will come into play. And so on.

To achieve this, assign a monotonically increasing or random priority to each operator in the deployment or replicaset:

```
kopf run --priority=$RANDOM ...
```

Or:

```
import kopf
import random
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.peering.priority = random.randint(0, 32767)
```

`$RANDOM` is a bash feature (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 unique priorities.

You can also use the pod's IP address in its numeric form as the priority, or any other source of integers.

32.7 Stealth keep-alive

Every few seconds (60 by default), the operator sends a keep-alive update to the chosen peering object, showing that it is still functioning. Other operators will notice this and decide whether to pause or resume.

The operator also logs keep-alive activity. This can be distracting. To disable it:

```
import kopf
import random
from typing import Any

@kopf.on.startup()
def configure(settings: kopf.OperatorSettings, **_: Any) -> None:
    settings.peering.stealth = True
```

There is no equivalent CLI option for that.

Note that this only affects logging. The keep-alive is still sent regardless.

CHAPTER

THIRTYTHREE

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`.

33.1 Scripting options

`-m, --module`
: A semantic equivalent of `python -m` — specifies which importable modules to import on startup.

33.2 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: logs all asyncio internals as well 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, the top-level key under which to place the object-identifying information, such as its name, namespace, etc.

33.3 Scope options

`-n, --namespace`
: Serve this namespace, or all namespaces matching the pattern (or excluded from patterns). This 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.

33.4 Probing options

--liveness

The endpoint on which 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

33.5 Peering options

--standalone

Disable peering and 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*), this 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

The priority to use for the operator. The operator with the highest priority wins the peering competition and handles the resources.

The default priority is `0`; *--dev* sets it to `666`.

See also

Peering

33.6 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

THIRTYFOUR

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.

34.1 Handled objects

Kopf provides some tools to report arbitrary information for the handled objects as Kubernetes events:

```
import kopf
from typing import Any

@kopf.on.create('kopfexamples')
def create_fn(body: kopf.Body, **_: Any) -> None:
    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 Python's `logging`:

```
import kopf
from typing import Any

@kopf.on.create('kopfexamples')
def create_fn(body: kopf.Body, **_: Any) -> None:
    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
```

34.2 Other objects

Events can also be attached to other objects, not only those currently being handled (and not necessarily even their children):

```python
import kopf
import kubernetes
from typing import Any

@kopf.on.create('kopfexamples')
def create_fn(name: str, namespace: str | None, uid: str, **_: Any) -> None:

    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 reported information should be treated as short-term only.

34.3 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.

The primary purpose of this is to prevent "event explosions" when handling core v1 events, which would create new core v1 events, causing more handling, and 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 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

THIRTYFIVE

HIERARCHIES

One of the most common operator patterns is to create child 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, or iterable containing several objects.

35.1 Labels

To label the resources to be created, use *kopf.label()*:

```
@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    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, omit the labels or set them to `None` (note that this is not the same as an empty dict `{}` — which is equivalent to doing nothing):

```
@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    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 any of the requested labels already exist, they will not be overwritten. To overwrite labels, use `forced=True`:

```
@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    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'}}}]
```

35.2 Nested labels

For some resources, such as `Job` or `Deployment`, additional fields must be modified to affect the doubly-nested children (`Pod` in this case).

To do this, their nested fields must be listed in a `nested=[...]` iterable. If there is only one nested field, it can be passed directly as `nested='...'`.

If the nested structures are absent in the target resources, they are skipped and no labels are added. Labels are added only to pre-existing structures:

```
@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    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 root-level resources, i.e. they are expected to have the `metadata` structure already, or it will be added automatically.

Nested resources are labelled *in addition* to the target resources. To label only the nested resources without the root resource, pass them directly to the function (e.g., `kopf.label(obj['spec']['template'], ...)`).

35.3 Owner references

Kubernetes natively supports 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 automatically, and no additional handlers are needed.

The `owner` is a dict containing the fields `apiVersion`, `kind`, `metadata.name`, and `metadata.uid` (other fields are ignored). This is usually the *body* from the handler keyword arguments, but you can construct your own dict or obtain one from a 3rd-party client library.

To set the ownership, use *kopf.append_owner_reference()*. To remove the ownership, use *kopf.remove_owner_reference()*:

```
owner = {'apiVersion': 'v1', 'kind': 'Pod', 'metadata': {'name': 'pod1', 'uid': '123...'}
↪}
kopf.append_owner_reference(objs, owner)
kopf.remove_owner_reference(objs, owner)
```

To add or remove ownership of the specified resource(s) by the resource currently being processed, omit the explicit owner argument or set it to `None`:

```
@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    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 that is not a controller or does 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.

35.4 Names

It is common to name child resources after the parent resource: either exactly as the parent, or with a random suffix.

To assign a name to resource(s), use `kopf.harmonize_naming()`. If the resource already has its `metadata.name` field set, that name will be used. If it does not, the specified name will be used. This 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 common mode when there are multiple child resources of the same kind. To ensure an 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 currently being processed, 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(**_: Any) -> None:
    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 only be known after the resource is created:

```
@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    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 approaches are commonly used for parent resources that orchestrate multiple child resources of the same kind (e.g., pods in a deployment).

35.5 Namespaces

Typically, child resources are expected to be created in the same namespace as their parent (unless there are strong reasons to do otherwise).

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 currently being processed, omit the namespace or set it to `None`:

```
@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    objs = [{'kind': 'Job'}, {'kind': 'Deployment'}]
    kopf.adjust_namespace(objs, forced=True)
    print(objs)
    # [{'kind': 'Job', 'metadata': {'namespace': 'default'}},
    #  {'kind': 'Deployment', 'metadata': {'namespace': 'default'}}]
```

35.6 Adopting

All of the above can be done in a single call with *kopf.adopt()*; the `forced`, `strict`, and `nested` flags are passed to all functions that support them:

```
@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    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'}}}]
```

35.7 3rd-party libraries

All described methods support resource-related classes from selected libraries in the same way as native Python dictionaries (or any mutable mappings). Currently, these are pykube-ng (classes based on `pykube.objects.APIObject`) and kubernetes client (resource models from `kubernetes.client.models`).

```
import kopf
import pykube
from typing import Any

@kopf.on.create('KopfExample')
def create_fn(**_: Any) -> None:
    api = pykube.HTTPClient(pykube.KubeConfig.from_env())
    pod = pykube.objects.Pod(api, {})
    kopf.adopt(pod)
```

```
import kopf
import kubernetes.client
from typing import Any

@kopf.on.create('KopfExample')
```

(continues on next page)

(continued from previous page)

```
def create_fn(**_: Any) -> None:
    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

THIRTYSIX

OPERATOR TESTING

Kopf provides some tools for testing Kopf-based operators via the *kopf.testing* module (requires explicit importing).

36.1 Background runner

kopf.testing.KopfRunner runs an arbitrary operator in the background while the original testing thread performs 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.output
    assert 'Deleted, really deleted' in runner.output
```

> **Note**
>
> The operator runs against the currently authenticated cluster — the same as if it were executed with `kopf run`.

36.2 Mock server

KMock is a supplementary project for running a local mock server for any HTTP API, and for the Kubernetes API in particular — with extended support for Kubernetes API endpoints, resource discovery, and implicit in-memory object persistence.

Use KMock when you need a very lightweight simulation of the Kubernetes API without deploying a full Kubernetes cluster, for example when migrating to/from Kopf.

```
import kmock
import requests

def test_object_patching(kmock: kmock.KubernetesEmulator) -> None:
    kmock.objects['kopf.dev/v1/kopfexamples', 'ns1', 'name1'] = {'spec': 123}
    requests.patch(str(kmock.url) + '/kopf.dev/v1/namespaces/ns1/name1', json={'spec': 456})
    assert len(kmock.requests) == 1
    assert kmock.requests[0].method == 'patch'
    assert kmock.objects['kopf.dev/v1/kopfexamples', 'ns1', 'name1'] == {'spec': 456}
```

KMock's detailed documentation is outside the scope of Kopf's documentation. The project and its documentation can be found at:

- https://kmock.readthedocs.io/
- https://github.com/nolar/kmock
- https://pypi.org/project/kmock/

CHAPTER

THIRTYSEVEN

EMBEDDING

Kopf is designed to be embeddable into other applications that require watching Kubernetes resources (custom or built-in) and handling their 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.

37.1 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 kopf
import threading
from typing import Any

@kopf.on.create('kopfexamples')
def create_fn(**_: Any) -> None:
    pass

def kopf_thread() -> None:
    asyncio.run(kopf.operator())

def main() -> None:
    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 outside the main thread, it cannot set OS signal handlers, so the developer should implement termination themselves (cancelling an operator task is enough).

37.2 Manual orchestration

Alternatively, a developer can orchestrate the operator's tasks and sub-tasks themselves. The example above is equivalent to the following:

```
def kopf_thread() -> None:
    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 required, of the following:

```
def kopf_thread() -> None:
    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 used properly. Specifically, `asyncio.run()` creates and finalises a new event loop for a single call. Multiple calls cannot share the same coroutines and tasks. To make multiple calls, either create a new event loop or get the event loop of the current asyncio _context_ (by default, that of the current thread). See more on asyncio event loops and _contexts_ in Asyncio Policies.

37.3 Custom event loops

Kopf can run in any AsyncIO-compatible event loop. For example, uvloop claims to be 2x–2.5x faster than asyncio. To run Kopf in uvloop, call it this way:

```
import kopf
import uvloop

def main() -> None:
    loop = uvloop.EventLoopPolicy().get_event_loop()
    loop.run(kopf.operator())
```

Or this way:

```
import kopf
import uvloop

def main() -> None:
    kopf.run(loop=uvloop.EventLoopPolicy().new_event_loop())
```

Or this way:

```
import kopf
import uvloop

def main() -> None:
    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`**) uses uvloop by default if it is installed. To disable this implicit behavior, 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 never implicitly activates custom event loops when called from the code, only from the CLI.

37.4 Multiple operators

Kopf can handle multiple resources at a time, so a single instance is sufficient for most cases. However, it may sometimes be necessary to run multiple isolated operators in the same process.

It should be safe to run multiple operators in multiple isolated event loops. Although Kopf's routines use global state, all such global state is stored in `contextvars` containers with values isolated per-loop and per-task.

```
import asyncio
import kopf
import threading
from typing import Any

registry = kopf.OperatorRegistry()

@kopf.on.create('kopfexamples', registry=registry)
def create_fn(**_: Any) -> None:
    pass

def kopf_thread() -> None:
    asyncio.run(kopf.operator(
        registry=registry,
    ))

def main() -> None:
    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 avoid cancelling tasks that existed before the operator's startup, but these cannot be applied to tasks spawned later due to asyncio implementation details.

CHAPTER

THIRTYEIGHT

DOCKER IMAGE

Kopf provides pre-built Docker images on the GitHub Container Registry (GHCR) with all extras pre-installed. These images are intended for quick experimentation and ad-hoc operator development. For production, it is recommended to build your own image with `pip install kopf` (see *Deployment*).

The images are available at:

```
ghcr.io/nolar/kopf
```

Each image includes the `kopf` CLI, the `full-auth` extra (`pykube-ng` and `kubernetes` client libraries), `uvloop` for better async performance, and the `dev` extra (`oscrypto`, `certbuilder`, `certvalidator`, `pyngrok`) for development convenience.

38.1 Image variants

Images are published for each release in several variants:

- **slim** (default) — based on Debian, larger but with broader compatibility.
- **alpine** — based on Alpine Linux, smaller but may have issues with some native dependencies.

Both variants are built for `linux/amd64` and `linux/arm64` platforms.

38.2 Image tags

For a release such as `1.43.0` built with Python 3.14 (the default), the following tags are available:

- `ghcr.io/nolar/kopf:latest` — the latest release with the default Python version and the default variant (slim).
- `ghcr.io/nolar/kopf:1.43.0` — a specific patch release.
- `ghcr.io/nolar/kopf:1.43` — the latest patch within a minor release.
- `ghcr.io/nolar/kopf:v1` — the latest release within a major version.

To pin a specific Python version or variant, use the extended tag format:

- `ghcr.io/nolar/kopf:1.43.0-python3.14-alpine`
- `ghcr.io/nolar/kopf:1.43-python3.14-slim`
- `ghcr.io/nolar/kopf:v1-python3.14`

Tags without a variant suffix (e.g. `1.43-python3.14`) point to the slim variant. Tags without a Python version (e.g. `1.43.0`) point to the default Python version.

38.3 Kubeconfig security

Mounting `~/.kube/config` directly into a container is risky: the default kubeconfig often contains access tokens or client certificates for multiple clusters, contexts, and service accounts. A container only needs access to one cluster in one context.

For safer local development, create a minified copy that contains only the current context:

```
kubectl config view --minify --flatten > dev.kubeconfig
```

Review the resulting file to ensure it contains only what you expect. All examples in this guide mount this minified config rather than the full `~/.kube/config`. Remember to regenerate the file when switching contexts or after token rotation.

38.4 Quick start

The simplest way to run an operator is to mount a single Python file at `/app/main.py` inside the container. The entrypoint will auto-detect it and run `kopf run -v /app/main.py`:

```
kubectl config view --minify --flatten > dev.kubeconfig
docker run --rm -it \
    -v ./handler.py:/app/main.py:ro \
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf
```

The `dev.kubeconfig` mount gives the operator access to the Kubernetes cluster (see *Kubeconfig security* above for how to create it).

38.5 Running a specific file

To run an operator from a custom path, pass the `run` command with the path explicitly:

```
kubectl config view --minify --flatten > dev.kubeconfig
docker run --rm -it \
    -v ./src:/src:ro \
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf run -v /src/handler.py
```

38.6 Extra dependencies

If the operator needs additional Python packages, the entrypoint supports two mechanisms for automatic installation at startup.

38.6.1 requirements.txt

Mount a `requirements.txt` file at `/app/requirements.txt`. The entrypoint will run `pip install -r /app/requirements.txt` before starting the operator:

```
kubectl config view --minify --flatten > dev.kubeconfig
docker run --rm -it \
    -v ./handler.py:/app/main.py:ro \
    -v ./requirements.txt:/app/requirements.txt:ro \
```

(continues on next page)

(continued from previous page)

```
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf
```

38.6.2 pyproject.toml

Mount the entire project directory at `/app/`. If a `pyproject.toml` is found, the entrypoint will run `pip install -e /app/` to install the project and its dependencies (i.e., as an editable project from the source directory):

```
kubectl config view --minify --flatten > dev.kubeconfig
docker run --rm -it \
    -v .:/app:rw \
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf
```

The main script should be `main.py` for auto-start. If not `main.py`, add the CLI arguments to run the custom modules or files:

```
kubectl config view --minify --flatten > dev.kubeconfig
docker run --rm -it \
    -v .:/app:rw \
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf run -m myoperator.main
```

Note

Note the `:rw` (read-write) mode on the `/app` directory — `pip` needs it for building the package locally before installing it.

38.7 Passing CLI options

Any arguments passed to the container are forwarded directly to the `kopf` CLI. For example, to run in verbose mode with a specific namespace:

```
kubectl config view --minify --flatten > dev.kubeconfig
docker run --rm -it \
    -v ./handler.py:/app/main.py:ro \
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf run /app/main.py --verbose --namespace=default
```

To see all available CLI options:

```
docker run --rm ghcr.io/nolar/kopf run --help
```

> **See also**
>
> *Command-line options* for the full list of command-line options.

38.8 Local clusters

When the Kubernetes API server runs on the host machine (e.g. k3d, k3s, kind, minikube, or Docker Desktop Kubernetes), the container cannot reach it at `127.0.0.1` because that address refers to the container itself, not the host.

The simplest solution is to use host networking, which shares the host's network stack with the container:

```
kubectl config view --minify --flatten > dev.kubeconfig
docker run --rm -it --network host \
    -v ./handler.py:/app/main.py:ro \
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf
```

With `--network host`, the kubeconfig's `127.0.0.1` addresses work as-is. This mode is supported on Linux natively, and on macOS via Docker Desktop and OrbStack.

If host networking is not an option, rewrite the server address in the kubeconfig to use `host.docker.internal` — a special hostname that resolves to the host machine in Docker Desktop and OrbStack:

```
kubectl config view --minify --flatten | \
    sed 's|127\.0\.0\.1|host.docker.internal|' > dev.kubeconfig

docker run --rm -it \
    -v ./handler.py:/app/main.py:ro \
    -v ./dev.kubeconfig:/root/.kube/config:ro \
    ghcr.io/nolar/kopf
```

On Linux without Docker Desktop, add `--add-host host.docker.internal:host-gateway` to make the hostname resolve to the host.

38.9 Using with Docker Compose

The image can be used in a Docker Compose setup for local development. Prepare the kubeconfig first:

```
kubectl config view --minify --flatten > dev.kubeconfig
```

```
services:
  operator:
    image: ghcr.io/nolar/kopf
    volumes:
      - ./handler.py:/app/main.py:ro
      - ./dev.kubeconfig:/root/.kube/config:ro
```

To target a local cluster (k3d, kind, etc.), use host networking:

```
kubectl config view --minify --flatten > dev.kubeconfig
```

```
services:
  operator:
    image: ghcr.io/nolar/kopf
    network_mode: host
    volumes:
      - ./handler.py:/app/main.py:ro
      - ./dev.kubeconfig:/root/.kube/config:ro
```

38.10 Building your own image

For production deployments, it is recommended to build a custom image rather than relying on the pre-built one. This avoids startup-time dependency installation, ensures reproducible builds, and keeps the image minimal:

```
FROM python:3.14
RUN pip install kopf
COPY handler.py /src/handler.py
CMD ["kopf", "run", "/src/handler.py", "--verbose"]
```

See also

Deployment for the full deployment guide, including RBAC and Kubernetes Deployment manifests.

CHAPTER

THIRTYNINE

DEPLOYMENT

Kopf can be run outside the cluster, as long as the environment is authenticated to access the Kubernetes API. Normally, however, operators are deployed directly into the cluster.

39.1 Docker image

First, the operator must be packaged as a Docker image with Python 3.10 or newer:

Listing 1: Dockerfile

```
FROM python:3.14
RUN pip install kopf
ADD . /src
CMD kopf run /src/handlers.py --verbose
```

Build and push it to a repository of your choice. Here, we use DockerHub (with the personal account "nolar" — replace it with your own name or namespace; you may also want to add version tags instead of the implied "latest"):

```
docker build -t nolar/kopf-operator .
docker push nolar/kopf-operator
```

See also

Read the DockerHub documentation for instructions on pushing and pulling Docker images.

39.2 Cluster deployment

The best way to deploy the operator to the cluster is via the Deployment object: it will be kept alive automatically, and upgrades will be applied properly on redeployment.

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
```

Note that there is only one replica. Keep it that way. If two or more operators run in the cluster for the same objects, they will collide with each other and the consequences are unpredictable. During pod restarts, only one pod should be running at a time as well: 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 typical web application examples), since the operator does not listen for incoming connections but only makes outgoing calls to the Kubernetes API.

39.3 RBAC

The pod where the operator runs must have permissions to access and manipulate objects, both domain-specific and 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 this, **RBAC**__ (Role-Based Access Control) can be used and attached to the operator's pod via a service account.

__: https://kubernetes.io/docs/reference/access-authn-authz/rbac/

Here is an example of what an RBAC config should look like (remove the parts that are not needed: e.g. the cluster roles and bindings for a 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]
```

(continues on next page)

(continued from previous page)

```
  # Application: other resources it produces and manipulates.
  # 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
```

Note that service accounts are always namespace-scoped. There are no cluster-wide service accounts. They must be created in the same namespace where the operator will run (even if it is going to serve the whole cluster).

CHAPTER

FORTY

CONTINUITY

40.1 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*).

40.2 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 operator's 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.

40.3 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 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

FORTYONE

IDEMPOTENCE

Kopf provides tools to make the handlers idempotent.

The *kopf.register()* function and the *kopf.subhandler()* decorator allow scheduling arbitrary sub-handlers for execution in the current cycle.

kopf.execute() coroutine executes arbitrary sub-handlers directly in the place of invocation, and returns when all of them 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
from typing import Any

@kopf.on.create('kopfexamples')
async def create(spec: kopf.Spec, namespace: str | None, **_: Any) -> None:
    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: int, **_: Any) -> None:
    if retry < 2:
        raise kopf.TemporaryError("Not ready yet.", delay=10)

async def create_b(retry: int, **_: Any) -> None:
    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. This repeats every ~10 seconds until both 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, even though `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 already succeeded, and the record of this is stored on the object.

This approach can be used to perform operations that need 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

FORTYTWO

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 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 synchronized according to the operator's concepts.

Kopf only provides a few ways and tools for achieving this easily.

42.1 Edge-based triggering

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. This is edge-based triggering (oversimplified).

This, however, fully excludes the *actual state* from the consideration, thus breaking the whole idea of reconciliation of the *actual state*.

It might be good for some simplistic operators that do not have *actual state* at all, and only care about the *desired state* as declared by the resources.

Yet, both high-level and low-level handlers are sufficient to keep track of the *desired state*, and react when it changes, assuming they already have the information on the *actual state*.

See also

Handlers

To keep track of the *actual state*, Kopf offers several ways:

42.2 Regularly scheduled timers

Timers are triggered on a regular schedule, regardless of whether anything changes or does not change in the resource itself. You can use timers to verify both the resource's body, and the state of other related resources through API calls, update the original resource's status to keep track of the *actual state*, and even bring the *actual state* to the *desired state*.

The little downside is that timers produce logs on every triggering, which can be noisy, especially if triggered often. Also, such an operator will not be very responsive to the changes in the *actual/desired states* (only as responsive as the timer's interval defines it).

See also

Timers

42.3 Permanently running daemons

Daemons are long-running activities or background tasks dedicated to each individual resource (object). They are typically an infinite cycle of the same operation running until the resource is deleted (daemon is stopped).

The benefit of daemons over timers is that daemons do not log too much, since they do not exit the function normally.

Besides, daemons can naturally use long-polling operations, which block until something changes in the remote system, and react immediately once it changes with no extra delays or polling intervals.

See also

Daemons

42.4 Level-based triggering

In Kubernetes, level-based triggering is the core concept of reconciliation. It implies that there is an *actual state* and a *desired state*. The latter usually sits in `spec`, while the former is calculated — it can come from inside the same Kubernetes cluster (children resources), other clusters, or other non-Kubernetes systems.

As a generic pattern, Kopf recommends implementing such level-based triggering and reconciliation the following way:

- Keep a timer or a daemon to regularly calculate the *actual state*, and store the result into the status stanza as one or several fields.
- For local Kubernetes resources as the *actual state*, use *In-memory indexing* instead of talking to the cluster API, in order to reduce the API load.
- Add on-field, or on-update/create handlers, or a low-level event handler for both the *actual state* and the *desired state* fields and react accordingly by bringing the actual state to the desired state.

An example for the in-cluster calculated *actual state* — this is not a full example (lacks wordy API calls for pods creation/termination), but you can get the overall idea:

```
import kopf
import random
from typing import Any

# Keep in-memory index of children resources, so that we avoid API calls doing the same.
@kopf.index('pods', labels={'parent-kex': kopf.PRESENT})
def kex_pods(body: kopf.Body, name: str, **_: Any) -> Any:
    parent_name = body.metadata.labels['parent-kex']
    return {parent_name: name}

# Regularly calculate and save the *actual state* from an in-memory index.
# If an in-memory index is absent, redesign this to make API calls to get the same data.
@kopf.timer('kopfexamples', interval=10)
def calculate_actual_state(name: str, kex_pods: kopf.Index, patch: kopf.Patch, **_: Any)␣
↪-> None:
```

(continues on next page)

(continued from previous page)

```
    actual_pods = kex_pods.get(name, [])
    patch.status['replicas'] = len(actual_pods)

# React to changes in either the *desired* or *actual* states, and reconcile them.
@kopf.on.event('kopfexamples')
def react_on_state_changes(body: kopf.Body, name: str, **_: Any) -> None:
    actual_replicas = body.status.get('replicas', 0)
    desired_replicas = body.spec.get('replicas', 1)
    delta = desired_replicas - actual_replicas
    if delta > 0:
        print(f"Spawn {delta} new pods with labels: {{'parent-kex': {name!r}}}.")
    if delta < 0:
        running_pods = kex_pods.get(name, [])
        pods_to_terminate = random.sample(running_pods, k=min(-delta, len(running_pods))
        print(f"Terminate {-delta} random pods: {pods_to_terminate}")
```

Time-based polling works well for both in-cluster and external *actual states*, and is in fact the only option for external *actual states* from third-party APIs.

For immediate reaction instead of polling, turn this timer into a daemon, introduce a global operator-scoped condition (e.g., an `asyncio.Condition`) stored in *In-memory containers* on operator startup, await it in the daemon of the parent resource, and notify it in the indexers of the children resources (mind the synchronisation: the index changes slightly after the indexer exits).

CHAPTER

FORTYTHREE

TIPS & TRICKS

43.1 Excluding handlers forever

Both successful executions and permanent errors of change-detecting handlers only exclude those 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 any previous permanent error.

The same applies to daemons: they will be spawned on the next operator restart (assuming 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 annotation filters (or the equivalent for labels or arbitrary fields with `when=` callback filtering):

```
import kopf
from typing import Any

@kopf.on.update('kopfexamples', annotations={'update-fn-never-again': kopf.ABSENT})
def update_fn(patch: kopf.Patch, **_: Any) -> None:
    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: kopf.Patch, **_: Any) -> None:
    patch.metadata.annotations['monitor-never-again'] = 'yes'
```

Such a never-again exclusion may be implemented as a built-in Kopf feature one day, but for now it is only available when implemented explicitly as shown above.

CHAPTER

FORTYFOUR

TROUBLESHOOTING

44.1 `kubectl` freezes on object deletion

This can happen if the operator is down at the moment of deletion.

The operator adds finalizers to 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 *actual deletion* by Kubernetes.

If the object must 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

FORTYFIVE

VISION

Kubernetes has become a de facto standard for 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 most widely used and easy-to-learn programming languages: Python.

But Kopf does not go too far in abstracting the Kubernetes internals away: it avoids introducing extra entities and control 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

FORTYSIX

NAMING

Kopf is an abbreviation either for **K**ubernetes **O**perator **P**ythonic **F**ramework, or for **K**ubernetes **OP**erator **F**ramework — whichever you prefer.

"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

FORTYSEVEN

CRITIQUES

> Critique is a constructive, detailed analysis aimed at improvement, focusing on both strengths and weaknesses. Conversely, criticism is often subjective, judgmental, and focused on finding faults, typically aimed at disapproval. While a critique encourages growth, criticism is often destructive. /Google on "critique vs. criticism"/

Kopf has several known design-level flaws. They are listed and addressed here to help you make the best decision on how to build your operator.

47.1 Python is slow and resource-greedy

Python is an interpreted language. Among other things, this means it consumes more memory at runtime and is overall slower than compiled languages such as Go. The Kubernetes ecosystem uses Go as its primary language, making it a natural and fair baseline for comparing operator performance.

Some reports (source lost) say that rewriting an operator from Python to Go reduced memory usage from 300 MB to 30 MB, i.e. 10x. These concerns are valid, and if memory usage is a critical concern, you should indeed use Go.

Internal measurements using `memray` and `memory_profiler` show that the operator's data structures consume barely any memory. With a no-op operator (no meaningful domain logic; no Kubernetes API clients installed) and 1,000 resources handled in an artificial setup, roughly 50% of memory went to imported Python modules, and 30% went to Python's TCP connections and SSL contexts for the API calls; only 20% was the operator runtime. With the official `kubernetes` client installed, modules took 75% of memory (the client alone took 60%), 15–20% went to TCP/SSL, and 10% to the operator. Switching from CPython to PyPy (which is officially supported by Kopf) gave no benefit.

To the best of my knowledge, nothing can be done about this. It is a focus of the Python community worldwide, and people are making efforts to improve Python's performance on many fronts.

However, Python is known not for its runtime performance but for its expressive power and ease of use, especially in the early stages of a product. It is a language for quick prototyping and for building minimally viable products rapidly. And since it is one of the most widely used languages, there is clearly demand for that simplicity.

Kopf follows the same paradigm: quickly starting operators and writing small operators for ad-hoc tasks. Kopf's roadmap includes work to improve performance and prepare for high-load tasks and large clusters, but it will never match Go-based operators in terms of resource usage — that is not the goal. Expressive power remains the primary value and the main goal.

47.2 Level-based vs. edge-based triggering

Another critique claims that Kopf suggests some bad practices in terms of handling resources: edge-based triggering instead of level-based triggering.

It is easier to explain with an example.

Imagine you have an operator with a resource containing the `spec.replicas` field, which is originally 3. You then change it to 2, then back to 3, while only 1 replica is actually running at the moment.

In level-based triggering, the change sequence 3→2→3 should not concern the operator at all. What matters is that you have 1 replica running, so the operator's effort should go toward bringing the actual state (1) to the desired state (2 or 3), i.e. adding 2 more replicas.

Kopf, on the other hand, focuses on the change (edge-based triggering) from 3 to 2 and back to 3 until it is applied. Kopf has no concept of "actual state" as a baseline for comparison, unless the operator developer implements it.

This is a fair critique, and I, as the author, fully admit this flaw. Nevertheless, I would like to offer a counter-argument.

Kopf's low-level handlers (on-event, indexes, daemons/timers) — those that do not track state — are direct equivalents of the Go/Kubernetes operator concepts used in level-based triggering. If you limit yourself to those, you can implement any reconciliation technique or state machine similar to Go-based operators.

Kopf's high-level handlers (on-creation/update/deletion) — those that do track state and calculate diffs — are intended for ease of use with ad-hoc operators. They are an add-on on top of what is available in Go/Kubernetes frameworks, in line with the stated goal of making it easy to express and prototype ad-hoc operators from scratch.

Indeed, the mere existence of these high-level handlers suggests and encourages the "bad practices" of event-driven operator design and edge-based triggering. But the decision always rests with the developers of the actual operators and which trade-offs they are willing or unwilling to make.

Developers aiming for high-grade Kubernetes-native operators built on the best practices of level-based triggering and reconciliation should learn the subtle differences between these concepts (edge- vs. level-based triggering) and design their operators accordingly from the very beginning. See *Reconciliation* for an example of level-based triggering with a calculated "actual state".

Kopf will support both approaches indefinitely, in line with its goals (ease of use, quick prototyping, ad-hoc solutions). Better reconciliation approaches may emerge later, connecting an "actual state" (calculated) to a "desired state" (usually `spec`). There are no specific timelines or plans.

CHAPTER

FORTYEIGHT

ALTERNATIVES

48.1 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 is it 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.

Kopf also allows writing *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 domain logic in any language due to the language-agnostic nature of HTTP APIs.

48.2 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 in Python 3 and allows writing 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 suitable if the scripts are written in shell/bash. Writing complex domain logic in bash can be troublesome.

Scripts in other languages, such as Python, are supported but require internal infrastructure logic to parse the input, render the output, and 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.

48.3 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 aforementioned 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 CoreOS's 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 Kubernetes is just one special case of such a domain logic, but operators could also be used to manage applications outside Kubernetes (via their APIs), or to implement 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

FORTYNINE

DEVELOPMENT STATUS

Kopf is production-ready and stable (semantic v1). Major bugs are fixed ASAP (there were none for a long time). Minor bugs are fixed as time and energy permit, or a workaround is provided.

There is no active development of **new major** functionality for Kopf — the whole idea of a framework for operators is fully expressed and implemented, I have nothing more to add. This piece of art is finished. (This might change.)

Minor feature requests can be implemented from time to time. Maintenance for new versions of Python and Kubernetes is performed regularly. Some internal optimizations are planned, such as a reduced memory footprint, high-load readiness, or agentic friendliness — but they will be backwards-compatible (no semantic v2 with breaking changes on the horizon).

CHAPTER

FIFTY

IMPRESSUM & DATENSCHUTZ

For the Impressum & Datenschutz documents, see the main website:

- https://kopf.dev/impressum
- https://kopf.dev/datenschutz

CHAPTER

FIFTYONE

MINIKUBE

To develop the framework and 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 it does not, or if you have multiple clusters, activate it explicitly:

```
kubectl config get-contexts
kubectl config current-context
kubectl config use-context minikube
```

To clean up minikube (and release CPU, RAM, and disk resources):

```
minikube stop
minikube delete
```

See also

For even more information, read the Minikube installation manual.

CHAPTER

FIFTYTWO

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 pass.
- Wait for a review by the project maintainers.

52.1 Git workflow

Kopf uses a Git Forking Workflow. This means all development should happen in individual forks, not in 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 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

52.2 Git conventions

The more rules you have, the less they are followed.

Kopf tries to avoid 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 prefixes or suffixes in commit messages or PR titles for issues or change types. In general, keep the git log clean — it will later go into the changelogs.
- Sign off your commits for DCO (see below).

No other rules.

52.3 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, posted at http://developercertificate.org/.

This is a developer's certification that they have the right to submit the patch for inclusion in 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 written manually or added with `git commit -s`. If you contribute often, you can automate this in Kopf's repo with a Git hook.

52.4 Code style

Common sense is the best code formatter. Blend your code into the surrounding code's style.

Kopf does not use and will never use strict code formatters (at least until they acquire common sense and context awareness). When in doubt, adhere to PEP 8 and the Google Python Style Guide.

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, and layered module checks, run:

```
pre-commit run
```

52.5 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 example, with k3s/k3d on macOS; Kind and Minikube are also fine):

```
brew install k3d
k3d cluster create
pytest --only-e2e
```

If that is not possible, create a draft PR instead, check the GitHub Actions results for unit and functional tests, fix as needed, and promote the draft PR to a full PR once everything is ready.

52.6 Reviews

If possible, reference the issue the PR addresses in the PR's body. You can use one of the existing or closed issues that best matches your topic.

PRs can be reviewed and commented on by anyone, but can be approved only by the project maintainers.

CHAPTER

FIFTYTHREE

ARCHITECTURE

53.1 Layered layout

The framework is organized into several layers, which are themselves layered. Higher-level layers and modules can import the lower-level ones, but not vice versa. import-linter checks and enforces the layering.

53.1.1 Root

At the topmost level, the framework consists of cogs, core, kits, and user-facing modules.

`kopf`, `kopf.on`, and `kopf.testing` are the public interface that can be imported by operator developers. Only these public modules carry public guarantees on names and signatures. Everything else is an implementation detail.

The internal modules are intentionally hidden (by underscore naming) to discourage taking dependencies on implementation details that may change without notice.

`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, cogs are fully independent of each other and of the rest of the framework — to the point that they could be extracted as separate libraries (in theory, if anyone needed it).

`core` is the main functionality used by a Kopf-based operator. It sets the operators in 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 specific scenarios and settings. The framework itself does not use them.

53.1.2 Cogs

`helpers` are system-level or language-enhancing adapters: for example, hostname identification, dynamic Python module importing, and integrations with third-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 domain.

`structs` are data structures and type declarations for Kubernetes models: resource kinds, selectors, bodies and their parts (specs, statuses, etc.), admission reviews, and so on. This also includes some specialised structures, such as authentication credentials — also abstracted from the framework even if the clients and their authentication are replaced.

`configs` are mostly settings, 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, this is based on aiohttp; previously, it used the official Kubernetes client library and pykube-ng. Over time, the entire

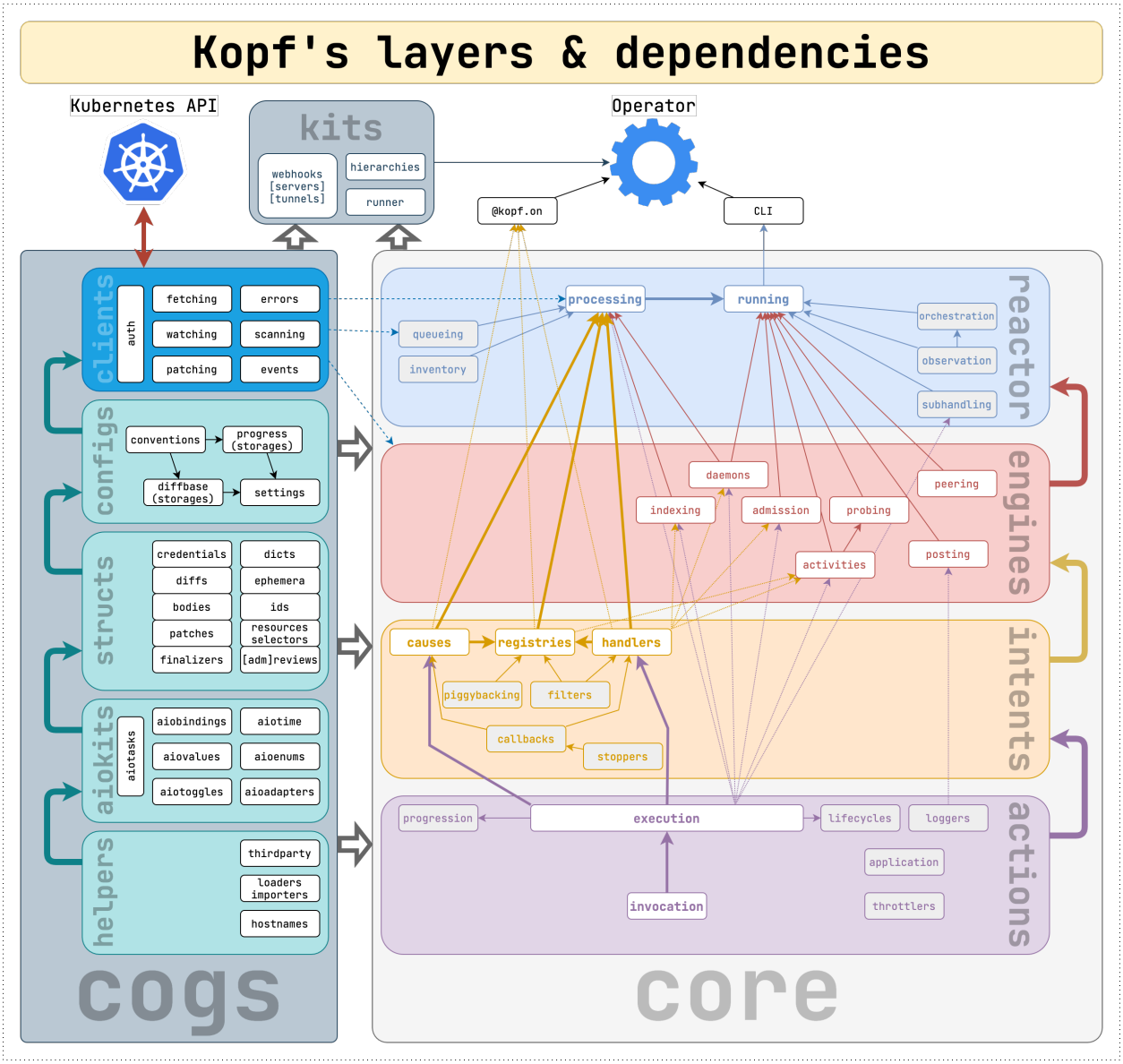

Fig. 1: The figure shows only the essential module dependencies, not all of them. Cross-layer dependencies represent all the many other imports.

client implementation can be replaced with another — while keeping the signatures for the rest of the framework. Only the clients are allowed to talk to the Kubernetes API.

53.1.3 Core

`actions` is the lowest level in the core (but not in the framework). It defines how functions and handlers are invoked, which ones specifically, how their errors are handled and retried (if at all), and how the function results and patches are applied to the cluster, and so on.

`intents` are mostly data structures that store the declared handlers of the operators, plus some logic to select and 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). Examples include daemons and timers, validating/mutating admission requests, in-memory indexing, operator activities (authentication, probing, etc.), peering, and Kubernetes `kind: Event` delayed posting.

`reactor` is the topmost layer in the framework. It defines the entry points for the CLI and operator embedding (see *Embedding*) and implements task orchestration for all engines and internal machinery. The reactor also observes the cluster for resources and namespaces, and dynamically spawns and stops tasks to serve them.

53.1.4 Kits

`hierarchies` are helper functions to manage hierarchies of Kubernetes objects: labelling them, adding and removing owner references, name generation, and so on. They support raw Python dicts as well as 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 when running locally on a developer's machine (see *Admission control*).

`runner` is a helper that runs an operator as a Python context manager, mostly useful for testing (see *Operator testing*).

CHAPTER

FIFTYFOUR

KOPF PACKAGE

The main Kopf module for all the exported functions and classes.

kopf.**subhandler**(** (Keyword-only parameters separator (PEP 3102))*, *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 effectively syntactic sugar to look like all other handlers:

```
import kopf

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

    for task in spec.get('tasks', []):

        @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-loop.

Return type
Callable[[ChangingFn], ChangingFn]

Parameters

- **id** (*str | None*)
- **param** (*Any | None*)
- **errors** (ErrorsMode | *None*)
- **timeout** (*float | None*)
- **retries** (*int | None*)
- **backoff** (*float | None*)

- **labels** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **old** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **new** (*Any | MetaFilterToken | MetaFilterFn | None*)

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:

```
import kopf

@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 effectively equivalent to:

```
import kopf

@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
ChangingFn

Parameters

- **fn** (*ChangingFn*)
- **id** (*str | None*)
- **param** (*Any | None*)
- **errors** (*ErrorsMode | None*)
- **timeout** (*float | None*)
- **retries** (*int | None*)
- **backoff** (*float | None*)

- **labels** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)

async kopf.**execute**(*, *fns=None*, *handlers=None*, *registry=None*, *lifecycle=None*, *cause=None*)

Execute the handlers in an isolated lifecycle.

This function is a public entry point 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[ChangingFn] | None*)
- **handlers** (*Iterable[ChangingHandler] | None*)
- **registry** (*ChangingRegistry | None*)
- **lifecycle** (*LifeCycleFn | None*)
- **cause** (*Cause | None*)

kopf.**daemon**(*arg1=None*, *arg2=None*, *arg3=None*, */ (Positional-only parameter separator (PEP 570))*, *, *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[[DaemonFn], DaemonFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | DelayFn | None*)
- **cancellation_backoff** (*float | None*)
- **cancellation_timeout** (*float | None*)
- **cancellation_polling** (*float | None*)
- **labels** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (OperatorRegistry | *None*)

kopf.**timer**(*arg1=None*, *arg2=None*, *arg3=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[[TimerFn], TimerFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | DelayFn | None*)
- **sharp** (*bool | None*)
- **idle** (*float | None*)
- **labels** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (OperatorRegistry | *None*)

kopf.**index**(*arg1=None*, *arg2=None*, *arg3=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[[IndexingFn], IndexingFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **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**(**values*)

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*, *settings*, ***_*)

Return type
ConnectionInfo | None

Parameters

- **logger** (*Logger | LoggerAdapter*)
- **settings** (OperatorSettings)
- **_** (*Any*)

kopf.**login_via_client**(***, *logger*, *settings*, ***_*)

Return type
ConnectionInfo | None

Parameters

- **logger** (*Logger | LoggerAdapter*)
- **settings** (OperatorSettings)

- **_** (*Any*)

async kopf.**login_via_async_client**(*, *logger*, *settings*, **_)

Return type
ConnectionInfo | None

Parameters

- **logger** (*Logger | LoggerAdapter*)
- **settings** (OperatorSettings)
- **_** (*Any*)

kopf.**login_with_kubeconfig**(*, *settings*, **_)

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
ConnectionInfo | None

Parameters

- **settings** (OperatorSettings)
- **_** (*Any*)

kopf.**login_with_service_account**(*, *settings*, **_)

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
ConnectionInfo | None

Parameters

- **settings** (OperatorSettings)
- **_** (*Any*)

exception kopf.**LoginError**

Bases: Exception

Raised when the operator cannot login to the API.

class kopf.**ConnectionInfo**(*, *server*, *priority=0*, *default_namespace=None*, *expiration=None*, *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*, *proxy_url=None*, *trust_env=False*)

Bases: KubeContext

A single endpoint with specific credentials and connection flags to use.

Parameters

- **server** (*str*)
- **priority** (*int*)
- **default_namespace** (*str | None*)
- **expiration** (*datetime | None*)
- **ca_path** (*str | bytes | PathLike[str] | PathLike[bytes] | None*)
- **ca_data** (*str | bytes | None*)
- **insecure** (*bool | None*)
- **username** (*str | None*)
- **password** (*str | None*)
- **scheme** (*str | None*)
- **token** (*str | None*)
- **certificate_path** (*str | bytes | PathLike[str] | PathLike[bytes] | None*)
- **certificate_data** (*str | bytes | None*)
- **private_key_path** (*str | bytes | PathLike[str] | PathLike[bytes] | None*)
- **private_key_data** (*str | bytes | None*)
- **proxy_url** (*str | None*)
- **trust_env** (*bool*)

server: str

ca_path: str | bytes | PathLike[str] | PathLike[bytes] | None = None

ca_data: str | bytes | None = None

insecure: bool | None = None

username: str | None = None

password: str | None = None

scheme: str | None = None

token: str | None = None

certificate_path: str | bytes | PathLike[str] | PathLike[bytes] | None = None

certificate_data: str | bytes | None = None

private_key_path: str | bytes | PathLike[str] | PathLike[bytes] | None = None

private_key_data: str | bytes | None = None

default_namespace: str | None = None

proxy_url: str | None = None

trust_env: bool = False

priority: int = 0

expiration: datetime | None = None

as_aiohttp_basic_auth()

Make a basic auth for username/password, or None if absent.

Return type
BasicAuth | None

as_http_headers()

Make a dict with the Authorization header set to scheme+token, or an empty dict if there are no tokens or schemes.

Return type
dict[str, str]

as_ssl_context()

Make an SSL context with CA and client cert using Python's ssl.

> **Warning**
>
> It will store the *kopf.ConnectionInfo.certificate_data* and *kopf.ConnectionInfo.private_key_data* into temporary files for a brief moment of time until the SSL context is constructed, since Python's ssl cannot load them from memory.

Return type
SSLContext

class kopf.**AiohttpSession**(*, *server*, *priority=0*, *default_namespace=None*, *expiration=None*, *aiohttp_session*)

Bases: KubeContext

A custom aiohttp session to use instead of the built-in one.

See: *Custom HTTP sessions* for details.

Parameters

- **server** (*str*)
- **priority** (*int*)
- **default_namespace** (*str | None*)
- **expiration** (*datetime | None*)
- **aiohttp_session** (*ClientSession*)

aiohttp_session: ClientSession

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[Any] | Event | None*)
- **ready_flag** (*Future | Event | Future[Any] | 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 forcibly terminated too.

> **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 forcibly 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).

Return type
None

Parameters

- **root_tasks** (*Collection[Task]*)
- **ignored** (*Collection[Task]*)

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 effectively *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[Any] | Event | None*)
- **ready_flag** (*Future | Event | Future[Any] | 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 args. 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[Any] | Event | None*)
- **ready_flag** (*Future | Event | Future[Any] | 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] | APIObject | KubernetesModelSync | KubernetesModelAsync | Iterable[MutableMapping[Any, Any] | APIObject | KubernetesModelSync | KubernetesModelAsync]*)
- **owner** (Body | *None*)
- **forced** (*bool*)
- **strict** (*bool*)
- **nested** (*str | Iterable[str | tuple[str, ...] | list[str] | None] | 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] | APIObject | KubernetesModelSync | KubernetesModelAsync | Iterable[MutableMapping[Any, Any] | APIObject | KubernetesModelSync | KubernetesModelAsync]*)
- **labels** (*Mapping[str, str | None] | _UNSET*)
- **forced** (*bool*)
- **nested** (*str | Iterable[str | tuple[str, ...] | list[str] | None] | None*)
- **force** (*bool | None*)

kopf.**not_**(*fn*)

Return type
TypeVar(_FnT, WhenFilterFn, MetaFilterFn)

Parameters
fn (*_FnT*)

kopf.**all_**(*fns*)

Return type
TypeVar(_FnT, WhenFilterFn, MetaFilterFn)

Parameters
fns (*Collection[_FnT]*)

kopf.**any_**(*fns*)

Return type
TypeVar(_FnT, WhenFilterFn, MetaFilterFn)

Parameters
fns (*Collection[_FnT]*)

kopf.**none_**(*fns*)

Return type
TypeVar(_FnT, WhenFilterFn, MetaFilterFn)

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] | APIObject | KubernetesModelSync | KubernetesModelAsync | Iterable[MutableMapping[Any, Any] | APIObject | KubernetesModelSync | KubernetesModelAsync]*)
- **owner** (Body | *None*)
- **controller** (*bool | None*)
- **block_owner_deletion** (*bool | None*)

kopf.**remove_owner_reference**(*objs*, *owner=None*)

Remove an owner reference from 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] | APIObject | KubernetesModelSync | KubernetesModelAsync | Iterable[MutableMapping[Any, Any] | APIObject | KubernetesModelSync | KubernetesModelAsync]*)
- **owner** (Body | *None*)

class kopf.**ErrorsMode**(**values*)

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 and 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: str | None

name: str | None

path: str | None

port: int | None

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: str | None

url: str | None

service: *WebhookClientConfigService* | None

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*, *file_check_interval=60.0*)

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*)
- **file_check_interval** (*float*)

DEFAULT_HOST: str | None = None

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

file_check_interval: float

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*, *file_check_interval=60.0*)

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*)
- **file_check_interval** (*float*)

DEFAULT_HOST: str | None = '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*, *file_check_interval=60.0*)

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*)
- **file_check_interval** (*float*)

DEFAULT_HOST: str | None = 'host.minikube.internal'

class kopf.**WebhookDockerDesktopServer**(*, *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*, *file_check_interval=60.0*)

Bases: *WebhookServer*

A tunnel from inside of Docker Desktop 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 Docker Desktop 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*)
- **file_check_interval** (*float*)

DEFAULT_HOST: str | None = 'host.docker.internal'

class kopf.**WebhookNgrokTunnel**(*, *addr=None*, *port=None*, *path=None*, *token=None*, *region=None*, *binary=None*)

Bases: `WebhookContextManager`

Tunnel admission webhook requests 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 to a remote public URL.

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 providing 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[str] | None*)

addr: str | None

port: int | None

path: str | None

token: str | None

region: str | None

binary: str | PathLike[str] | 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*, *file_check_interval=60.0*)

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*.

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*)
- **file_check_interval** (*float*)

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 activate 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 & tunnels: 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: str | None

port: int | None

path: str | None

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>*, *queueing=<factory>*, *scanning=<factory>*, *admission=<factory>*, *execution=<factory>*, *background=<factory>*, *networking=<factory>*, *persistence=<factory>*)

Bases: object

Parameters

- **process** (*ProcessSettings*)
- **posting** (*PostingSettings*)
- **peering** (*PeeringSettings*)
- **watching** (*WatchingSettings*)
- **queueing** (*QueueingSettings*)
- **scanning** (*ScanningSettings*)
- **admission** (*AdmissionSettings*)
- **execution** (*ExecutionSettings*)
- **background** (*BackgroundSettings*)
- **networking** (*NetworkingSettings*)
- **persistence** (*PersistenceSettings*)

process: ProcessSettings

posting: PostingSettings

peering: PeeringSettings

watching: WatchingSettings

queueing: QueueingSettings

scanning: ScanningSettings

admission: AdmissionSettings

execution: ExecutionSettings

background: BackgroundSettings

networking: NetworkingSettings

persistence: PersistenceSettings

property batching: QueueingSettings

class kopf.**DiffBaseStorage**(*ignored_fields=None*)

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/

Parameters

ignored_fields (*Iterable[str | tuple[str, ...] | list[str] | None] | None*)

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 stanza 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[str | tuple[str, ...] | list[str] | None] | None*)

abstractmethod fetch(*, *body*)

Return type

BodyEssence | None

Parameters

body (Body)

abstractmethod 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'*, *ignored_fields=None*, *v1=True*)

Bases: StorageKeyFormingConvention, *DiffBaseStorage*

Parameters

- **prefix** (*str*)
- **key** (*str*)
- **ignored_fields** (*Iterable[str | tuple[str, ...] | list[str] | None] | None*)
- **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 stanza 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[str | tuple[str, ...] | list[str] | None] | None*)

fetch(*, *body*)

Return type
BodyEssence | None

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'*, *ignored_fields=None*)

Bases: *DiffBaseStorage*

Parameters

- **name** (*str*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **ignored_fields** (*Iterable[str | tuple[str, ...] | list[str] | None] | None*)

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 stanza 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[str | tuple[str, ...] | list[str] | None] | None*)

fetch(*, *body*)

Return type
BodyEssence | None

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 stanza 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[str | tuple[str, ...] | list[str] | None] | None*)

fetch(*, *body*)

Return type
BodyEssence | None

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: str | None

stopped: str | None

delayed: str | None

purpose: str | None

retries: int | None

success: bool | None

failure: bool | None

message: str | None

subrefs: Collection[HandlerId] | None

class kopf.**ProgressStorage**

Bases: StorageStanzaCleaner

Base class and an interface for all persistent states.

The state is persisted strictly 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: handlers should be persisted on their own, unrelated to other handlers, and never combined with 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.

abstractmethod fetch(*, *key*, *body*)

Return type
ProgressRecord | None

Parameters

- **key** (*HandlerId*)
- **body** (Body)

abstractmethod store(*, *key*, *record*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*)
- **record** (ProgressRecord)
- **body** (Body)
- **patch** (Patch)

abstractmethod purge(*, *key*, *body*, *patch*)

Return type
None

Parameters

- **key** (*HandlerId*)
- **body** (Body)
- **patch** (Patch)

abstractmethod touch(*, *body*, *patch*, *value*)

Return type
None

Parameters

- **body** (Body)

- **patch** (Patch)
- **value** (*str | None*)

abstractmethod 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
ProgressRecord | None

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** (*str | tuple[str, ...] | list[str] | None*)
- **touch_field** (*str | tuple[str, ...] | list[str] | None*)

property field: tuple[str, ...]

property touch_field: tuple[str, ...]

fetch(*, *key*, *body*)

Return type
ProgressRecord | None

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
ProgressRecord | None

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** (*str | tuple[str, ...] | list[str] | None*)
- **touch_key** (*str*)
- **touch_field** (*str | tuple[str, ...] | list[str] | None*)
- **prefix** (*str*)
- **v1** (*bool*)

- **verbose** (*bool*)

class kopf.**RawEvent**

Bases: TypedDict

type: Literal[None, 'ADDED', 'MODIFIED', 'DELETED', 'BOOKMARK']

object: *RawBody*

class kopf.**RawBody**

Bases: TypedDict

apiVersion: str

kind: str

metadata: RawMeta

spec: dict[str, Any]

status: dict[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

property name: str

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 (RawBody | BodyEssence)

property metadata: *Meta*

property meta: *Meta*

property spec: *Spec*

property status: *Status*

class kopf.BodyEssence

Bases: TypedDict

metadata: MetaEssence

spec: dict[str, Any]

status: dict[str, Any]

class kopf.ObjectReference

Bases: TypedDict

apiVersion: str

kind: str

namespace: str | None

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 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**(**values*)

Bases: str, Enum

ADD = 'add'

CHANGE = 'change'

REMOVE = 'remove'

class kopf.**Reason**(**values*)

Bases: str, Enum

CREATE = 'create'

UPDATE = 'update'

DELETE = 'delete'

RESUME = 'resume'

NOOP = 'noop'

FREE = 'free'

GONE = 'gone'

class kopf.**Patch**(*src=None*, */*, *body=None*, *fns=()*)

Bases: dict[str, Any]

Parameters

- **src** (*dict[str, Any] | None*)
- **body** (*Body | None*)
- **fns** (*Iterable[Callable[[RawBody], None]]*)

clear()

Remove all items from the dict.

Return type

None

property fns: list[Callable[[*RawBody*], None]]

property metadata: MetaPatch

property meta: MetaPatch

property spec: SpecPatch

property status: StatusPatch

as_json_patch(*body=None*)

Build a list of JSON-patch ops for the changes & transformations.

As a reference resource body, either the argument is used (if provided), or the original resource body. But the reference body is mandatory — the patch calculates the differences relative to the reference body.

Some changes might disappear from the list if they are useless (no-op): e.g., setting a key to `None` to delete it when it is already absent; or setting the key to a value which is already in the resource body.

Return type
list[JSONPatchItem]

Parameters
body (Body | RawBody | *None*)

class kopf.DaemonStoppingReason(**values*)

Bases: Flag

A reason or combination of reasons for a 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 for 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: `str | None = None`

The resource's kind (as in YAML files); e.g. `"Pod"`, `"KopfExample"`.

singular: `str | None = 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: `bool | None = 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** (*dict[str, str] | None*)

54.1 Submodules

54.1.1 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[Any] | Event | None*)
- **stop_flag** (*Future | Event | Future[Any] | Event | None*)
- **vault** (*Vault | None*)
- **registry** (*OperatorRegistry | None*)
- **settings** (*OperatorSettings | None*)
- **loop** (*AbstractEventLoop | None*)

ready_flag: Future | Event | Future[Any] | Event | None = None

stop_flag: Future | Event | Future[Any] | Event | None = None

vault: Vault | None = None

registry: *OperatorRegistry* | None = None

settings: *OperatorSettings* | None = None

loop: AbstractEventLoop | None = None

kopf.cli.**logging_options**(*fn*)

A decorator to configure logging in all commands the same way.

Return type
Callable[..., Any]

Parameters
fn (*Callable[[...], Any]*)

54.1.2 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[[ActivityFn], ActivityFn]

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[[ActivityFn], ActivityFn]

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[[ActivityFn], ActivityFn]

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[[ActivityFn], ActivityFn]

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**(*arg1=None, arg2=None, arg3=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[[WebhookFn], WebhookFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (*OperatorRegistry | None*)

kopf.on.**mutate**(*arg1=None*, *arg2=None*, *arg3=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`[[WebhookFn], WebhookFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (*OperatorRegistry | None*)

kopf.on.**resume**(*arg1=None*, *arg2=None*, *arg3=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 resuming objects on operator (re)start.

Return type
`Callable`[[ChangingFn], ChangingFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)

- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (*OperatorRegistry | None*)

kopf.on.**create**(*arg1=None*, *arg2=None*, *arg3=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[[ChangingFn], ChangingFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (*OperatorRegistry | None*)

kopf.on.**update**(*arg1=None*, *arg2=None*, *arg3=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`[[ChangingFn], ChangingFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **old** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **new** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (*OperatorRegistry | None*)

kopf.on.**delete**(*arg1=None*, *arg2=None*, *arg3=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[[ChangingFn], ChangingFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (*OperatorRegistry | None*)

kopf.on.**field**(*arg1=None*, *arg2=None*, *arg3=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[[ChangingFn], ChangingFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **old** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **new** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (*OperatorRegistry | None*)

kopf.on.**index**(*arg1=None*, *arg2=None*, *arg3=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`[[IndexingFn], IndexingFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (OperatorRegistry | *None*)

kopf.on.**event**(*arg1=None*, *arg2=None*, *arg3=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 silently observing all events.

Return type
Callable[[WatchingFn], WatchingFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)

- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **registry** (OperatorRegistry | *None*)

kopf.on.**daemon**(*arg1=None*, *arg2=None*, *arg3=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[[DaemonFn], DaemonFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | DelayFn | None*)
- **cancellation_backoff** (*float | None*)
- **cancellation_timeout** (*float | None*)
- **cancellation_polling** (*float | None*)
- **labels** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)

- **registry** (OperatorRegistry | None)

kopf.on.**timer**(*arg1=None*, *arg2=None*, *arg3=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[[TimerFn], TimerFn]

Parameters

- **arg1** (*str | Marker | Callable[[Resource], bool] | None*)
- **arg2** (*str | Marker | None*)
- **arg3** (*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 | DelayFn | None*)
- **sharp** (*bool | None*)
- **idle** (*float | None*)
- **labels** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **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 effectively syntactic sugar to look like all other handlers:

```
import kopf

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

    for task in spec.get('tasks', []):

        @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-loop.

Return type
`Callable`[[ChangingFn], ChangingFn]

Parameters

- **id** (*str | None*)
- **param** (*Any | None*)
- **errors** (*ErrorsMode | None*)
- **timeout** (*float | None*)
- **retries** (*int | None*)
- **backoff** (*float | None*)
- **labels** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)
- **field** (*str | tuple[str, ...] | list[str] | None*)
- **value** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **old** (*Any | MetaFilterToken | MetaFilterFn | None*)
- **new** (*Any | MetaFilterToken | MetaFilterFn | None*)

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:

```
import kopf

@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 effectively equivalent to:

```
import kopf

@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
ChangingFn

Parameters

- **fn** (*ChangingFn*)
- **id** (*str | None*)
- **param** (*Any | None*)
- **errors** (ErrorsMode | *None*)
- **timeout** (*float | None*)
- **retries** (*int | None*)
- **backoff** (*float | None*)
- **labels** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **annotations** (*Mapping[str, str | MetaFilterToken | MetaFilterFn] | None*)
- **when** (*WhenFilterFn | None*)

54.1.3 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

def test_operator():
    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.output
```

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` 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, args) are lost.

Parameters

- **args** (*Any*)
- **reraise** (*bool*)
- **timeout** (*float | None*)
- **registry** (OperatorRegistry | *None*)
- **settings** (OperatorSettings | *None*)
- **kwargs** (*Any*)

property future: Future[Result]

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

FIFTYFIVE

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

Q

R

S

T

U

V

W