Krake

Release 1.0.0

Feb 28, 2023

Contents:

CHAPTER 1

Quickstart

A simple introduction to Krake can be found on the README on the official GitLab repository. You can find there all the requirements and the different steps for the installation, as well as some basic commands and initiatory explanations.

CHAPTER 2

User Documentation

2.1 Rok documentation

The Rok utility has a command line interface with a few specific commands, that can be added one after the other to refer to specific elements. The general syntax is

```
rok <api> <resource> <operation> <parameters>
```

The separate elements are:

`api` element: The name of the Krake API used. Different APIs are present to handle different kind of resources. Example: `kube` for the Kubernetes API of Krake.

`resource` element: The name of the resource managed. Each API holds one or several resources it can handle. Example: `cluster` for the Krake **Clusters**, which correspond to Kubernetes clusters.

`operation` element: The verb used for the operation to apply. For instance `list` can be used to get all instances of one kind of resource, while `delete` can be used to remove a resource.

`parameters` element: The specific argument for the current operation. For instance, the `-o | --output` argument change the format of the response.

A few examples:

```
$ rok kube <...>  # handle the kubernetes API resources

$ rok kube app <...>  # handle the Application resources of the Kubernetes API

# Register a cluster with the Kubernetes API using the minikube.yaml kubeconfig
$ rok kube cluster register --kubeconfig ../minikube.yaml

# Create a cluster with the Kubernetes API using the tosca.yaml manifest
$ rok kube cluster create --file ../tosca.yaml test-cluster
```

2.1.1 The `kube` API

This API can be used to manage Kubernetes clusters and start, update and delete applications on them, through Krake.

Base command: `rok kube <...>`

The Cluster resource: `cluster`

This resource manages Krake **Cluster** resources, which needs to be registered or created on Krake to be used. It corresponds to a cluster on Kubernetes.

Base command: `rok kube cluster <...>`

register Add an existing cluster to the Kubernetes clusters registered in Krake on a specified namespace. Example:

```
rok kube cluster register -k <path_to_kubeconfig_file>
```

`-k | --kubeconfig`: the path to the kubeconfig file that refers to the cluster to register.

`-n | --namespace` (optional): The namespace to which the Cluster has to be added. If none is given, the user namespace is selected.

`-c | --context` (optional): The name of the context to use from the kubeconfig file. Only one context can be chosen at a time. If not context is specified, the current context of the kubeconfig file is chosen.

`--global-metric` (optional): The name and weight of of a global cluster metric in the form: `<name> <weight>`. Can be specified multiple times.

`-m | --metric` (optional): The name and weight of a cluster metric in the form: `<name> <weight>`. Can be specified multiple times.

`-l | --label` (optional): The key and the value of a cluster label in the form: `<key>=<value>`. Can be specified multiple times.

`-R | --custom-resource` (optional): The name of custom resources definition in the form: `<plural>.<group>` which is supported by the cluster. Can be specified multiple times.

create Add a new cluster to the Kubernetes clusters registered in Krake on a specified namespace. Example:

```
rok kube cluster create <cluster_name> -f <path_to_tosca_template>
```

`name`: The name of the new Cluster, as stored by Krake (can be arbitrary). The same name cannot be used twice in the same namespace.

`-f | --file`: The path to the TOSCA template file that describes the desired Cluster.

`-n | --namespace` (optional): The namespace to which the Cluster has to be added. If none is given, the user namespace is selected.

`--global-metric` (optional): The name and weight of a global cluster metric in the form: `<name> <weight>`. Can be specified multiple times.

`-m | --metric` (optional): The name and weight of a cluster metric in the form: `<name> <weight>`. Can be specified multiple times.

`-l | --label` (optional): The key and the value of a cluster label in the form: `<key>=<value>`. Can be specified multiple times.

`-R | --custom-resource` (optional): The name of custom resources definition in the form: `<plural>.<group>` which is supported by the cluster. Can be specified multiple times.

-L | --cloud-label-constraint (optional): The name and value of a constraint for labels of the cloud in the form: `<label> expression <value>`. The cluster will be deployed only on the cloud that matches the given label constraint. Can be specified multiple times, see *Constraints*.

-M | --cloud-metric-constraint (optional): The name and value of a constraint for metrics of the cloud in the form: `<label> expression <value>`. The cluster will be deployed only on the cloud that matches the given metric constraint. Can be specified multiple times, see *Constraints*.

--backoff (optional): multiplier applied to backoff_delay between attempts. default: 1 (no backoff)

`backoff_delay` (optional): delay [s] between attempts. default: 1

`backoff_limit` (optional): a maximal number of attempts. If the attempt to handle the cluster failed, it will transfer to the Cluster State DEGRADED, instead of directly going into the State OFFLINE. Default: -1 (infinite) default: -1 (infinite)

list List all Cluster of a namespace.

-n | --namespace (optional): The namespace from which the Clusters have to be listed. If none is given, the user namespace is selected.

get Request information about a specific Cluster.

name: The name of the Cluster to fetch.

-n | --namespace (optional): The namespace from which the Clusters have to be retrieved. If none is given, the user namespace is selected.

update Request a change of the current state of an existing Cluster.

name: The name of the Cluster to update.

-k | --kubeconfig (optional): The path to the kubeconfig file that describes the Cluster with the updated fields.

-f | --file (optional): The path to the TOSCA template file that describes the desired Cluster with the updated fields.

-n | --namespace (optional): The namespace from which the Clusters have to be taken. If none is given, the user namespace is selected.

-c | --context (optional): The name of the context to use from the kubeconfig file. Only one context can be chosen at a time. If not context is specified, the current context of the kubeconfig file is chosen.

--global-metric (optional): The name and weight of a global cluster metric in the form: `<name> <weight>`. Can be specified multiple times.

-m | --metric (optional): The name and weight of a cluster metric in the form: `<name> <weight>`. Can be specified multiple times.

-l | --label (optional): The key and the value of a cluster label in the form: `<key>=<value>`. Can be specified multiple times.

-R | --custom-resource (optional): The name of custom resources definition in the form: `<plural>.<group>` which is supported by the cluster. Can be specified multiple times.

-L | --cloud-label-constraint (optional): The name and value of a constraint for labels of the cloud in the form: `<label> expression <value>`. The cluster will be deployed only on the cloud that matches the given label constraint. Can be specified multiple times, see *Constraints*.

-M | --cloud-metric-constraint (optional): The name and value of a constraint for metrics of the cloud in the form: `<label> expression <value>`. The cluster will be deployed only on the cloud that matches the given metric constraint. Can be specified multiple times, see *Constraints*.

--backoff (optional): multiplier applied to backoff_delay between attempts. default: 1 (no backoff)

`backoff_delay` (optional): delay [s] between attempts. default: 1

`backoff_limit` (optional): a maximal number of attempts, default: -1 (infinite)

delete Request the deletion of a specific Cluster from a namespace.

`-n | --namespace` (optional): The namespace from which the Cluster have to be deleted. If none is given, the user namespace is selected.

`--force` (optional): Force the deletion of resources directly from the Krake Database.

The Application resource: `app`

This resource manages Krake **Applications** resources, which need to be registered on Krake to be managed. It corresponds to a Kubernetes resource.

Tip: Krake is able to manage applications that are described by Kubernetes manifests files as well as by TOSCA templates or CSAR archives, see *TOSCA*.

Base command: `rok kube app <...>`

create Add a new Application to the ones registered on Krake on a specified namespace. Example:

```
rok kube app create <application_name> -f <path_to_manifest_or_path_to_tosca_
↪template>
```

`name`: The name of the new Application, as stored by Krake (can be arbitrary). The same name cannot be used twice in the same namespace.

`-f | --file`: The path to the manifest file or the TOSCA template file that describes the new Application.

`-u | --url`: The URL of the TOSCA template file or the CSAR archive that describes the new Application.

`-O | --observer_schema` (optional): The path to the custom observer schema file, specifying the fields of the Kubernetes resources defined in the manifest file which should be observed. If none is given, all fields defined in the manifest file are observed. The custom observer schema could be used even when the application is described by the TOSCA template or CSAR archive.

`-n | --namespace` (optional): The namespace to which the Application has to be added. If none is given, the user namespace is selected.

`--hook-complete` (optional): The complete hook, which allows an Application to send a completion signal to the API.

`--hook-shutdown` (optional): The shutdown hook, which allows the graceful shutdown of the Application. Can have an additional timeout value after the argument.

`-R | --cluster-resource-constraint` (optional): The name of custom resources definition constraint in form: `<plural>.<group>`. The application will be deployed only on the clusters with given custom definition support. Can be specified multiple times.

`-L | --cluster-label-constraint` (optional): The name and value of a constraint for labels of the cluster in the form: `<label> expression <value>`. The application will be deployed only on the cluster that matches the given label constraint. Can be specified multiple times, see *Constraints*.

`-M | --cluster-metric-constraint` (optional): The name and value of a constraint for metrics of the cluster in the form: `<label> expression <value>`. The application will be deployed only on the cluster that matches the given metric constraint. Can be specified multiple times, see *Constraints*.

--backoff (optional): multiplier applied to backoff_delay between attempts to handle the application. default: 1 (no backoff)

`backoff_delay` (optional): delay [s] between attempts to handle the application. default: 1

`backoff_limit` (optional): a maximal number of attempts to handle the application. If the attempt to handle the application failed, it will transfer to the Application State DEGRADED, instead of directly going into the State FAILED. Default: -1 (infinite)

list List all Applications of a namespace.

-n | --namespace (optional): The namespace from which the Applications have to be listed. If none is given, the user namespace is selected.

get Request information about a specific Application.

name: The name of the Application to fetch.

-n | --namespace (optional): The namespace from which the Applications have to be retrieved. If none is given, the user namespace is selected.

update Request a change of the current state of an existing Application.

name: The name of the Application to update.

-f | --file: The path to the manifest file or TOSCA template file that describes the Application with the updated fields.

-u | --url: The URL of the TOSCA template file or the CSAR archive that describes the Application with the updated fields.

-O | --observer_schema (optional): The path to the custom observer schema file, specifying the fields of the Kubernetes resources defined in the manifest file which should be observed. If none is given, the observer schema is not udpated. The custom observer schema could be used even when the application is described by the TOSCA template or CSAR archive.

-n | --namespace (optional): The namespace from which the Applications have to be taken. If none is given, the user namespace is selected.

--hook-complete (optional): The complete hook, which allows an Application to send a completion signal to the API.

--hook-shutdown (optional): The shutdown hook, which allows the graceful shutdown of the Application. Can have an additional timeout value after the argument.

-R | --cluster-resource-constraint (optional): The name of custom resources definition constraint in form: `<plural>.<group>`. The application will be deployed only on the clusters with given custom definition support. Can be specified multiple times.

-L | --cluster-label-constraint (optional): The name and value of a constraint for labels of the cluster in the form: `<label> expression <value>`. The application will be deployed only on the cluster that matches the given label constraint. Can be specified multiple times, see *Constraints*.

-M | --cluster-metric-constraint (optional): The name and value of a constraint for metrics of the cluster in the form: `<label> expression <value>`. The application will be deployed only on the cluster that matches the given metric constraint. Can be specified multiple times, see *Constraints*.

--backoff (optional): multiplier applied to backoff_delay between attempts. default: 1 (no backoff)

`backoff_delay` (optional): delay [s] between attempts. default: 1

`backoff_limit` (optional): a maximal number of attempts, default: -1 (infinite)

delete Request the deletion of a specific Application from a namespace.

name: The name of the Application to delete.

-n | --namespace (optional): The namespace from which the Application have to be deleted. If none is given, the user namespace is selected.

--force (optional): Force the deletion an Application directly from the Krake Database.

2.1.2 The `infra` API

This API can be used to manage the following infrastructure resources:

- GlobalInfrastructureProvider
- InfrastructureProvider
- GlobalCloud
- Cloud

Base command: `rok infra <...>`

The GlobalInfrastructureProvider resource: `globalinfrastructureprovider`

This resource manages Krake **GlobalInfrastructureProvider** non-namespaced resources, which needs to be registered on Krake to be used. It corresponds to an infrastructure provider software, that is able to deploy infrastructures (e.g. Virtual machines, Kubernetes clusters, etc.) on IaaS Cloud deployments (e.g. OpenStack, AWS, etc.).

Krake currently supports the following GlobalInfrastructureProvider software (types):

- IM (Infrastructure Manager) tool developed by the GRyCAP research group

Base command: `rok infra globalinfrastructureprovider <...>` Available aliases: - `rok infra gprovider <...>` - `rok infra gip <...>`

Note: The global resource is a non-namespaced resource that could be used by any (even namespaced) Krake resource. For example, the global infrastructure provider resource could be used by any cloud which needs to be managed by the infrastructure provider.

---

register Add a new GlobalInfrastructureProvider to the ones registered on Krake. Example:

```
rok infra gprovider register <provider_name> \
  --type <provider_type> \
  --url <provider_api_url> \
  --username <provider_api_username> \
  --password <provider_api_password>
```

name: The name of the new GlobalInfrastructureProvider, as stored by Krake (can be arbitrary). The same name cannot be used twice.

--type: The GlobalInfrastructureProvider type. Type of the infrastructure provider that will be registered on Krake. Currently, only IM infrastructure provider is supported, and valid type is: *im*.

--url: The GlobalInfrastructureProvider API url. Valid together with –type im.

--username (optional): The GlobalInfrastructureProvider API username. Valid together with –type im.

--password (optional): The GlobalInfrastructureProvider API password. Valid together with –type im.

--token (optional): The GlobalInfrastructureProvider API token. Valid together with –type im.

list List all GlobalInfrastructureProviders.

get Request information about a specific GlobalInfrastructureProvider.

`name`: The name of the GlobalInfrastructureProvider to fetch.

update Request a change of the current state of an existing GlobalInfrastructureProvider.

`name`: The name of the GlobalInfrastructureProvider to update.

`--url` (optional): The GlobalInfrastructureProvider API url to update. Valid together with –type im.

`--username` (optional): The GlobalInfrastructureProvider API username to update. Valid together with –type im.

`--password` (optional): The GlobalInfrastructureProvider API password to update. Valid together with –type im.

`--token` (optional): The GlobalInfrastructureProvider API token to update. Valid together with –type im.

delete Request the deletion of a specific GlobalInfrastructureProvider.

`name`: The name of the GlobalInfrastructureProvider to delete.

The InfrastructureProvider resource: `infrastructureprovider`

This resource manages Krake **InfrastructureProvider** namespaced resources, which needs to be registered on Krake to be used. It corresponds to an infrastructure provider software, that is able to deploy infrastructures (e.g. Virtual machines, Kubernetes clusters) on IaaS Cloud deployments.

Krake currently supports the following InfrastructureProvider software (types):

- IM (Infrastructure Manager) tool developed by the GRyCAP research group

Base command: `rok infra infrastructureprovider <...>`

Available aliases:

- `rok infra provider <...>`
- `rok infra ip <...>`

Note: This resource is a namespaced resource that could be used by the Krake resources from the same namespace. For example, the infrastructure provider resource could be used by any cloud which lives in the same namespace as the infrastructure provider.

---

register Add a new InfrastructureProvider to the ones registered on Krake. Example:

```
rok infra provider register <provider_name> \
  --type <provider_type> \
  --url <provider_api_url> \
  --username <provider_api_username> \
  --password <provider_api_password>
```

`name`: The name of the new InfrastructureProvider, as stored by Krake (can be arbitrary). The same name cannot be used twice in the same namespace.

`-n | --namespace` (optional): The namespace to which the InfrastructureProvider have to be added. If none is given, the user namespace is selected.

`--type`: The InfrastructureProvider type. Type of the infrastructure provider that will be registered on Krake. Currently, only IM infrastructure provider is supported, and valid type is: *im*.

--url: The InfrastructureProvider API url. Valid together with –type im.

--username (optional): The InfrastructureProvider API username. Valid together with –type im.

--password (optional): The InfrastructureProvider API password. Valid together with –type im.

--token (optional): The InfrastructureProvider API token. Valid together with –type im.

list List all InfrastructureProviders of a namespace.

-n | --namespace (optional): The namespace from which the InfrastructureProvider have to be listed. If none is given, the user namespace is selected.

get Request information about a specific InfrastructureProvider.

name: The name of the InfrastructureProvider to fetch.

-n | --namespace (optional): The namespace from which the InfrastructureProvider have to be retrieved. If none is given, the user namespace is selected.

update Request a change of the current state of an existing InfrastructureProvider.

name: The name of the InfrastructureProvider to update.

-n | --namespace (optional): The namespace from which the InfrastructureProvider have to be taken. If none is given, the user namespace is selected.

--url (optional): The InfrastructureProvider API url to update. Valid together with –type im.

--username (optional): The InfrastructureProvider API username to update. Valid together with –type im.

--password (optional): The InfrastructureProvider API password to update. Valid together with –type im.

--token (optional): The InfrastructureProvider API token to update. Valid together with –type im.

delete Request the deletion of a specific InfrastructureProvider from a namespace.

name: The name of the InfrastructureProvider to delete.

-n | --namespace (optional): The namespace from which the InfrastructureProvider have to be deleted. If none is given, the user namespace is selected.

The GlobalCloud resource: `globalcloud`

This resource manages Krake **GlobalCloud** non-namespaced resources, which needs to be registered on Krake to be used. It corresponds to an IaaS Cloud deployments (e.g. OpenStack, AWS, etc.) that will be managed by the infrastructure provider software. GlobalCloud resource could contain also metrics and labels, that could be used in cluster scheduling.

Krake currently supports the following GlobalCloud cloud software (types):

- OpenStack

Base command: `rok infra globalcloud <...>`

Available aliases:

- `rok infra gcloud <...>`
- `rok infra gc <...>`

Note: The global resource is a non-namespaced resource that could be used by any (even namespaced) Krake resource. For example, the global cloud resource could be used by any cluster which needs to be scheduled to some cloud.

---

register Add a new GlobalCloud to the ones registered on Krake. Example:

```
rok infra gcloud register <cloud_name> \
  --type <cloud_type> \
  --url <cloud_identity_service_url> \
  --username <cloud_username> \
  --password <cloud_password> \
  --project <cloud_project_name> \
  --global-infra-provider <global_infra_provider_name>
```

name: The name of the new GlobalCloud, as stored by Krake (can be arbitrary). The same name cannot be used twice.

--type: The GlobalCloud type. Type of the cloud that will be registered on Krake. Currently, only OpenStack cloud software is supported, and valid type is: *openstack*.

--url: URL to OpenStack identity service (Keystone). Valid together with –type openstack.

--username: Username or UUID of OpenStack user. Valid together with –type openstack.

--password: Password of OpenStack user. Valid together with –type openstack.

--project: Name or UUID of the OpenStack project. Valid together with –type openstack.

--global-infra-provider: Global infrastructure provider name for cloud management. Valid together with –type openstack.

--domain-name (optional): Domain name of the OpenStack user. Valid together with –type openstack.

--domain-id (optional): Domain ID of the OpenStack project. Valid together with –type openstack.

--global-metric (optional): The name and weight of a global cloud metric in form: `<name> <weight>`. Can be specified multiple times.

-l | --label (optional): The key and the value of cloud label in form: `<key>=<value>`. Can be specified multiple times.

list List all GlobalClouds.

get Request information about a specific GlobalCloud.

name: The name of the GlobalCloud to fetch.

update Request a change of the current state of an existing GlobalCloud.

name: The name of the GlobalCloud to update.

--url (optional): URL to OpenStack identity service (Keystone) to update. Valid together with –type openstack.

--username (optional): Username or UUID of OpenStack user to update. Valid together with –type openstack.

--password (optional): Password of OpenStack user to update. Valid together with –type openstack.

--project (optional): Name or UUID of the OpenStack project to update. Valid together with –type openstack.

--global-infra-provider (optional): Global infrastructure provider name for cloud management to update. Valid together with –type openstack.

--domain-name (optional): Domain name of the OpenStack user to update. Valid together with –type openstack.

--domain-id (optional): Domain ID of the OpenStack project to update. Valid together with –type openstack.

--global-metric (optional): The name and weight of cloud global metric in form: `<name> <weight>`. Can be specified multiple times.

-l | --label (optional): The key and the value of cloud label in form: `<key>=<value>`. Can be specified multiple times.

delete Request the deletion of a specific GlobalCloud.

name: The name of the GlobalCloud to delete.

The Cloud resource: `cloud`

This resource manages Krake **Cloud** namespaced resources, which needs to be registered on Krake to be used. It corresponds to an IaaS Cloud deployments (e.g. OpenStack, AWS, etc.) that will be managed by the infrastructure provider software. Cloud resource could contain also metrics and labels, that could be used in cluster scheduling.

Krake currently supports the following GlobalCloud cloud software (types):

- OpenStack

Base command: `rok infra cloud <...>`

Note: This resource is a namespaced resource that could be used by the Krake resources from the same namespace. For example, the cloud resource could be used by any cluster which lives in the same namespace as the cloud.

---

register Add a new Cloud to the ones registered on Krake. Example:

```
rok infra cloud register <cloud_name> \
  --type <cloud_type> \
  --url <cloud_identity_service_url> \
  --username <cloud_username> \
  --password <cloud_password> \
  --project <cloud_project_name> \
  --infra-provider <infra_provider_name>
```

name: The name of the new Cloud, as stored by Krake (can be arbitrary). The same name cannot be used twice in the same namespace.

-n | --namespace (optional): The namespace to which the Cloud have to be added. If none is given, the user namespace is selected.

--type: The Cloud type. Type of the cloud that will be registered on Krake. Currently, only OpenStack cloud software is supported, and valid type is: *openstack*.

--url: URL to OpenStack identity service (Keystone). Valid together with –type openstack.

--username: Username or UUID of OpenStack user. Valid together with –type openstack.

--password: Password of OpenStack user. Valid together with –type openstack.

--project: Name or UUID of the OpenStack project. Valid together with –type openstack.

--infra-provider (optional): Infrastructure provider name for cloud management. Valid together with –type openstack.

--global-infra-provider (optional): Global infrastructure provider name for cloud management to update. Valid together with –type openstack.

--domain-name (optional): Domain name of the OpenStack user. Valid together with –type openstack.

--domain-id (optional): Domain ID of the OpenStack project. Valid together with –type openstack.

--global-metric (optional): The name and weight of cloud global metric in form: <name> <weight>. Can be specified multiple times.

-m | --metric (optional): The name and weight of cloud metric in form: <name> <weight>. Can be specified multiple times.

-l | --label (optional): The key and the value of cloud label in form: <key>=<value>. Can be specified multiple times.

list List all Clouds of a namespace.

-n | --namespace (optional): The namespace from which the Cloud have to be listed. If none is given, the user namespace is selected.

get Request information about a specific Cloud.

name: The name of the Cloud to fetch.

-n | --namespace (optional): The namespace from which the Cloud have to be retrieved. If none is given, the user namespace is selected.

update Request a change of the current state of an existing Cloud.

name: The name of the Cloud to update.

-n | --namespace (optional): The namespace from which the Cloud have to be taken. If none is given, the user namespace is selected.

--url (optional): URL to OpenStack identity service (Keystone) to update. Valid together with –type openstack.

--username (optional): Username or UUID of OpenStack user to update. Valid together with –type openstack.

--password (optional): Password of OpenStack user to update. Valid together with –type openstack.

--project (optional): Name or UUID of the OpenStack project to update. Valid together with –type openstack.

--infra-provider (optional): Infrastructure provider name for cloud management to update. Valid together with –type openstack.

--global-infra-provider (optional): Global infrastructure provider name for cloud management to update. Valid together with –type openstack.

--domain-name (optional): Domain name of the OpenStack user to update. Valid together with –type openstack.

--domain-id (optional): Domain ID of the OpenStack project to update. Valid together with –type openstack.

--global-metric (optional): The name and weight of cloud global metric in form: <name> <weight>. Can be specified multiple times.

-m | --metric (optional): The name and weight of cloud metric in form: <name> <weight>. Can be specified multiple times.

-l | --label (optional): The key and the value of cloud label in form: <key>=<value>. Can be specified multiple times.

delete Request the deletion of a specific Cloud from a namespace.

name: The name of the Cloud to delete.

-n | --namespace (optional): The namespace from which the Cloud have to be deleted. If none is given, the user namespace is selected.

2.1.3 Common options

These options are common to all commands:

`-o | --output <format>` (optional): The format of the displayed response. Three are available: YAML: `yaml`, JSON: `json` or table: `table`.

2.1.4 Warnings

Warning messages are issued in situations where it is useful to alert the user of some condition in a Krake, which may exhibit errors or unexpected behavior. Warnings standard library is used, hence the warning messages could be filtered by `PYTHONWARNINGS` environment variable.

An example to disable all warnings:

```
$ PYTHONWARNINGS=ignore rok kube app create <...>
```

2.2 Configuration

This sections describes the configuration of Krake components and Rok. The different parameters, their value and role will be described here

Note: If an example value is specified for a parameter, it means this parameter has no default value in Krake.

2.2.1 Configuration file or command-line options

There are two different ways to configure Krake components:

- using the configuration files (also for Rok);
- using command-line options (only for Krake components).

Configuration files

There are 7 different configuration files:

- `api.yaml` for the Krake API;
- `scheduler.yaml` for the Scheduler as controller;
- `kubernetes_application.yaml` for the Kubernetes Application controller;
- `kubernetes_cluster.yaml` for the Kubernetes Cluster controller;
- `garbage_collection.yaml` for the Garbage Collector as controller;
- `infrastructure.yaml` for the Infrastructure controller;
- `rok.yaml` for the Rok utility.

For each one of them except `rok.yaml`, a template is present in the `config` directory. They end with the `.template` extension. For Rok, the template configuration file is in the main directory of Krake.

Generate configuration

From the templates, actual configuration files can be generated using the `krake_generate_config` script. The templates have parameters that can be overwritten by the script. It allows setting some parameters using command-line options. The arguments and available options are:

`<src_files> <src_files> ...<src_files>` (list of file paths) Positional arguments: the list of template files that will be used for generation.

`--dst` (path to a directory) Optional argument: the directory in which the generated files will be created. Default: `.` (current directory).

`--tls-enabled` If used, set the TLS support to enabled between all Krake components. By default, TLS is disabled.

`--cert-dir <cert_dir>` (path to a directory) Set the directory in which the certificates for the TLS communication should be stored. Default: `"tmp/pki"`.

`--allow-anonymous` If enabled, anonymous requests are accepted by the API. See *Authentication*. Disabled by default for the generation.

`--keystone-authentication-enabled` Enable the Keystone authentication as one of the authentication mechanisms. See *Authentication*. Disabled by default for the generation.

`--keystone-authentication-endpoint` Endpoint to connect to the keystone service. See *Authentication*. Default: `"http://localhost:5000/v3"`.

`--keycloak-authentication-enabled` Enable the Keycloak authentication as one of the authentication mechanisms. See *Authentication*. Disabled by default for the generation.

`--keycloak-authentication-endpoint` Endpoint to connect to the Keycloak service. See *Authentication*. Default: `"http://localhost:9080"`.

`--keycloak-authentication-realm` Keycloak realm to use on the provided endpoint. See *Authentication*. Default: `krake`.

`--static-authentication-enabled` Enable the static authentication as one of the authentication mechanisms. See *Authentication*. Disabled by default.

`--static-authentication-username` Name of the user that will authenticate through static authentication. See *Authentication*. Default: `"system:admin"`.

`--cors-origin` URL or wildcard for the 'Access-Control-Allow-Origin' of the CORS system on the API. Default: `*`.

`--authorization-mode` Authorization mode to use for the requests sent to the API. Only 'RBAC' should be used in production. See *Authorization*. Default: `always-allow`.

`--api-host <api_host>` (Address) Host that will be used to create the endpoint of the API for the controllers. Default: `"localhost"`.

`--api-port <api_port>` (integer) Port that will be used to create the endpoint of the API for the controllers.. Default: `8080`.

`--etcd-version <etcd_version>` (string) The etcd database version. Default: `v3.3.13`.

`--etcd-host <etcd_host>` (Address) Host for the API to use to connect to the etcd database. Default: `127.0.0.1`.

`--etcd-port <etcd_port>` (integer) Port for the API to use to connect to the etcd database. Default: `2379`.

`--etcd-port <etcd_port>` (integer) Peer port for the etcd endpoint. Default: `2380`.

--docs-problem-base-url <docs_problem_base_url> (string) URL of the problem documentation. Default: `https://rak-n-rok.readthedocs.io/projects/krake/en/latest/user/problem`.

--docker-daemon-mtu <docker_daemon_mtu> (integer) The Docker daemon MTU. Default: `1450`.

--worker-count <worker_count> (integer) Number of worker to start on the controller. Workers are the units that handle resources. Default: `5`.

--debounce <debounce> (float) For the controllers: the worker queue has a mechanism to delay a received state of a resource with a timer. A newer state received will then restart the timer. If a resource is updated a few times in one second, this mechanism prevents having to handle it each time by another component, and wait for the latest value. Default: `1.0`.

--reschedule-after Time in seconds after which a resource will be rescheduled. See *Scheduling*. Default: `60`.

--stickiness "Stickiness" weight to express migration overhead in the normalized ranking computation. See *Scheduling*. Default: `0.1`.

--poll-interval Time in seconds for the Infrastructure Controller to ask the infrastructure provider client again after a modification of a cluster. Default: `30`.

--complete-hook-user For the complete hook, set the name of the user that will be defined as CN of the generated certificates. See *Complete*. Default: `"system:complete-hook"`.

--complete-hook-cert-dest For the complete hook, set the path to the mounted directory, in which the certificates to communicate with the API will be stored. See *Complete*. Default: `"/etc/krake_cert"`.

--complete-hook-env-token For the complete hook, set the name of the environment variable that contain the value of the token, which will be given to the Application. See *Complete*. Default: `"KRAKE_COMPLETE_TOKEN"`.

--complete-hook-env-url For the complete hook, set the name of the environment variable that contain the URL of the Krake API, which will be given to the Application. See *Complete*. Default: `"KRAKE_COMPLETE_URL"`.

--external-endpoint (str) If set, replaces the value of the URL host and port of the endpoint given to the Applications which have the 'complete' hook enabled. See *Complete*.

--logging-level (str) To set the logging level of a controller. Default: `INFO`.

--logging-handler (str) To set the handler to use for logging. This lets one choose whether the logging messages should be printed to stdout or saved to a file. Options are 'console' and 'file'. Default: `console`.

-h, --help Display the help message and exit the script.

Examples

To create default configuration files for Krake, the following command can be used in the main directory:

```
krake_generate_config config/*template
```

This will create all Krake configuration files in the main directory of Krake.

To create default configuration files for Rok, the following command can be used in the main directory:

```
krake_generate_config rok.yaml.template
```

This will create the Rok configuration file in the main directory of Krake.

The two previous commands can be combined together to generate both Rok and Krake configuration files at the same time:

```
krake_generate_config config/*template rok.yaml.template
```

This will create Krake and Rok configuration files in the main directory of Krake.

To create a new configuration for the API on the `tmp` directory with a different etcd database endpoint, the following can be used:

```
krake_generate_config --dst /tmp config/api.yaml.template --etcd-host newhost.org --
↪etcd-port 1234
```

Command-line options

Apart from the configuration files, specific command-line options are available for the Krake components. They are created automatically from the configuration parameters. Nested options are generated by concatenating the names of section with dashes characters (`"-"`). For example, the `authentication.allow_anonymous` YAML element becomes the `--authentication-allow-anonymous` option.

There is one option for each parameter of the configuration, except the elements that are lists for the moment. Booleans are converted into optional flags.

2.2.2 Krake configuration

All configuration options for the Krake API are described here.

port (integer) This parameter defines the port to which the Krake API will listen to for incoming requests.

etcd This section defines the parameters to let the API communicate with the ETCD database.

host (string) Address of the database. Example: `127.0.0.1`

port (integer), default: `2379` Port to communicate with the database.

retry_transactions (int): Number of times a database transaction will be attempted again if it failed the first time due to concurrent write on the same resource.

tls This section defines the parameters needed for TLS support. If TLS is enabled, all other components and clients need TLS support to communicate with the API.

enabled (boolean) Activate or deactivate the TLS support. Example: `false`

cert (path) Set the path to the client certificate authority. Example: `tmp/pki/system:api-server.pem`

key (path) Set the path to the client certificate. Example: `tmp/pki/system:api-server-key.pem`

client_ca (path) Set the path to the client key. Example: `tmp/pki/ca.pem`

Authentication and authorization

authentication This section defines the method for authenticating users that connect to the API. Three methods are available: *keystone*, *keycloak* and *static*. A user not recognized can still send request if *anonymous* are allowed.

allow_anonymous (boolean), default: `false` Enable the "anonymous" user. Any request executed without a user being authenticated will be processed as user `system:anonymous`.

strategy This section describes the parameters for the methods of authentication.

keystone The Keystone service of OpenStack can be used as authentication method.

enabled (boolean) Set Keystone as authentication method. Example: `false`

endpoint (URL) Endpoint of the Keystone service. Example: `http://localhost:5000/v3`

keycloak The Keycloak service can be used as authentication method.

enabled (boolean) Set Keycloak as authentication method. Example: `false`

endpoint (URL) Endpoint of the Keycloak service. Example: `http://localhost:9080`

realm (str) Keycloak realm to use at the provided endpoint. Example: `krake`

static The user is set here, and the API will authenticate all requests as being sent by this user.

enabled (boolean) Set the static method as authentication method. Example: `true`

name (string) This is the name of the user that will be set as sending all requests. Example: `system`

cors-origin (string), default * For the CORS mechanism of Krake. Set the default allowed URL, which corresponds to the `Access-Control-Allow-Origin` response header.

authorization (enumeration) This parameter defines the mode for allowing users to perform specific actions (e.g. "create" or "delete" a resource). Three modes are available: `RBAC`, `always-allow`, `always-deny`.

2.2.3 Controllers configuration

The general configuration is the same for each controller. Additional parameters can be added for specific controllers, depending on the implementation. Here are the common parameters:

api_endpoint (URL) Address of the API to be reached by the current controller. Example: `http://localhost:8080`

debounce (float) For the worker queue of the controller: set the debounce time to delay the handling of a resource, and get any updated state in-between. Example `1.5`

tls This section defines the parameters needed for TLS support. If TLS support is enabled on the API, it needs to be enabled on the controllers to let them communicate with the API.

enabled (boolean) Activate or deactivate the TLS support. If the API uses only TLS, then this should be set to `true`. This has priority over the scheme given by *api_endpoint*. Example: `false`

client_ca (path) Set the path to the client certificate authority. Example: `./tmp/pki/ca.pem`

client_cert (path) Set the path to the client certificate. Example: `./tmp/pki/jc.pem`

client_key (path) Set the path to the client key. Example: `./tmp/pki/jc-key.pem`

Kubernetes application controller

Additional parameters, specific for the Kubernetes application controller:

hooks (string) All the parameters for the application hooks are described here. See also *Complete*.

complete (string) This section defines the parameters needed for the Application `complete` hook. If is not defined the Application `complete` hook is disabled.

hook_user (string) Name of the user that will be set as CN in the certificates generated for the hook. If RBAC is enabled, should match a `RoleBinding` for the `applications/complete` subresource. Example `system:complete-hook`

intermediate_src (path) Path to the certificate which will be used to sign new generated certificates for the hook. Not needed if TLS is not enabled. Example: `/etc/krake/certs/system:complete-signing.pem`

intermediate_key_src (path) Path to the key of the certificate which will be used to sign new generated certificates for the hook. Not needed if TLS is not enabled. Example: `/etc/krake/certs/system:complete-signing-key.pem`

cert_dest (path) Set the path to the certificate authority on the deployed Application. Example: `/etc/krake_cert`

env_token (string) Name of the environment variable, which stores Krake authentication token. Example: `KRAKE_COMPLETE_TOKEN`

env_url (string) Name of the environment variable, which stores Krake `complete` hook URL. Example: `KRAKE_COMPLETE_URL`

external_endpoint (URL, optional) If set, replaces the host and port in the value of environment variable in the Krake `complete` hook URL (the name of this variable is given by **env_url_**). By default, the value stored in the variable is the *api_endpoint*. Example: `https://krake.external.host:1234`.

shutdown (string) This section defines the parameters needed for the Application `shutdown` hook. If is not defined the Application `shutdown` hook is disabled.

hook_user (string) Name of the user that will be set as CN in the certificates generated for the hook. If RBAC is enabled, should match a `RoleBinding` for the `applications/shutdown` subresource. Example `system:shutdown-hook`

intermediate_src (path) Path to the certificate which will be used to sign new generated certificates for the hook. Not needed if TLS is not enabled. Example: `/etc/krake/certs/system:shutdown-signing.pem`

intermediate_key_src (path) Path to the key of the certificate which will be used to sign new generated certificates for the hook. Not needed if TLS is not enabled. Example: `/etc/krake/certs/system:shutdown-signing-key.pem`

cert_dest (path) Set the path to the certificate authority on the deployed Application. Example: `/etc/krake_cert`

env_token (string) Name of the environment variable, which stores Krake authentication token. Example: `KRAKE_SHUTDOWN_TOKEN`

env_url (string) Name of the environment variable, which stores Krake `shutdown` hook URL. Example: `KRAKE_SHUTDOWN_URL`

external_endpoint (URL, optional) If set, replaces the host and port in the value of environment variable in the Krake `shutdown` hook URL (the name of this variable is given by **env_url_**). By default, the value stored in the variable is the *api_endpoint*. Example: `https://krake.external.host:1234`.

Scheduler

Additional parameters, specific for the Scheduler:

reschedule_after (float): Number of seconds between the last update or rescheduling of a resource and the next rescheduling. Example: `60`

stickiness (float): Additional weight for the computation of the rank of the scheduler. It is added to the computation of the rank of the cluster on which a scheduled resource is actually running. It prevents migration from happening too frequently, and thus, represents the cost of migration. As the computation is done with normalized weights, the stickiness is advised to be between 0 and 1. Example: `0.1`.

Infrastructure controller

Additional parameters, specific for the Infrastructure controller:

poll_interval (float): Time in seconds for the Infrastructure Controller to ask the infrastructure provider client again after a modification of a cluster. Example: `30`.

2.2.4 Common configuration:

The following elements are common for all components of Krake except Rok.

Logging

log: This section is dedicated to the logging of the application. The syntax follows the one described for the Python logging module (`logging.config`). The content of this section will be given to this module for configuration.

2.2.5 Rok configuration

api_url (URL) Address of the Krake API to connect to. If the scheme given is incompatible with the *tls.enabled* parameter, it will be overwritten to match. Example: `http://localhost:8080`

user (string) The name of the user that will access the resources. Example: `john-doe`

tls This section defines the parameters needed for TLS support, which can be used to communicate with the API.

enabled (boolean) Activate or deactivate the TLS support. If the API uses only TLS, then this should be set to `true`. This has priority over the scheme given by *api_url*. Example: `false`

client_ca (path) Set the path to the client certificate authority. Example: `./tmp/pki/ca.pem`

client_cert (path) Set the path to the client certificate. Example: `./tmp/pki/jc.pem`

client_key (path) Set the path to the client key. Example: `./tmp/pki/jc-key.pem`

2.3 Custom Observer Schema

2.3.1 Purpose

When a user creates Kubernetes resources on a Kubernetes cluster via Krake, those resources are managed by Krake and should be "observed". That's the role of the Kubernetes Observer (see the dev/observers:Observers documentation). But what parts of the Kubernetes resources should be "observed" by Krake? The purpose of the Observer Schema is to provide a flexible mean for the Krake users to define which fields of the Kubernetes resources should be "observed" and which shouldn't.

When a field is "observed", every change to the value of this field made outside of Krake is reverted to the last known state of this field. When a field is not "observed", Krake doesn't act on external changes made to this field. This is needed to keep a consistent and predictable application state, especially since changes could also be done in the Kubernetes infrastructure or by the Kubernetes plane itself.

Note: The custom observer schema could be used even when the application is described by a TOSCA template or CSAR archive. Both file types are translated to Kubernetes manifests in Krake's Kubernetes application controller,

hence the custom observer schema file will be applied to the Kubernetes resources just like it happens during a "regular" workflow, when a Kubernetes manifest is used, see dev/tosca:TOSCA Workflow.

Note: As Kubernetes manages some fields of a Kubernetes resource (for instance the ResourceVersion), simply observing the entirety of a Kubernetes resource is not possible. This would lead to infinite reconciliation loops between Krake and Kubernetes, which is not a desirable state.

2.3.2 Format

Example

This basic example will be re-used at different part of this documentation.

Example of manifest file provided by the user:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echo-demo
  namespace: secondary
spec:
  selector:
    matchLabels:
      app: echo
  template:
    metadata:
      labels:
        app: echo
    spec:
      containers:
      - name: echo
        image: k8s.gcr.io/echoserver:1.10
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: echo-demo
  namespace: secondary
spec:
  type: NodePort
  selector:
    app: echo
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
```

Example of custom observer schema provided by the user.

```
---
apiVersion: v1
```

(continues on next page)

(continued from previous page)

```
kind: Service
metadata:
  name: echo-demo
  namespace: default
spec:
  selector:
    app: null
  ports:
  - port: null
    protocol: null
    targetPort: null
  - port: null
    protocol: null
    targetPort: null
  - observer_schema_list_min_length: 1
    observer_schema_list_max_length: 4
  sessionAffinity: null
```

Default observer schema

By default, all fields defined in `spec.manifest` are observed. All other fields are not observed. By defining a custom observer schema, the user is able to overwrite the default behavior and precisely define the observed fields.

In the example above, the user didn't specify a custom observer schema file for the `Deployment` resource. Therefore Krake will generate a default observer schema, and observe only the fields which are specified in the manifest file.

The result default observer schema for the `Deployment` resource is:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echo-demo
  namespace: secondary
spec:
  selector:
    matchLabels:
      app: null
  template:
    metadata:
      labels:
        app: null
    spec:
      containers:
      - name: null
        image: null
        ports:
        - containerPort: null
        - observer_schema_list_min_length: 1
          observer_schema_list_max_length: 1
      - observer_schema_list_min_length: 1
        observer_schema_list_max_length: 1
```

Resource identification

In order to identify which resource a schema is referring to, the `apiVersion`, `kind` and `name` need to be specified. Those fields are also the minimum fields a user can specify in order to observe a resource. As a result, and without additional fields to observe, the Kubernetes Observer will simply check the presence of a Kubernetes resource with this `apiVersion`, `kind` and `name`.

Example of a minimal observer schema for the `Service` resource:

```
---
apiVersion: v1
kind: Service
metadata:
  name: echo-demo
```

Note: The Kubernetes namespace key `metadata.namespace` is not mandatory, as it is not used in the identification of a resource in Krake. Indeed, its value is not always known at the creation of the application. It can depend from the Kubernetes cluster the application is scheduled to.

Please note that not all Kubernetes objects are in a namespace. Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. However, namespace resources are not themselves in a namespace. And low-level resources, such as nodes and persistentVolumes, are not in any namespace.

Therefore, Krake (by default) does not observe a Kubernetes namespace field.

Users may choose to add the `metadata.namespace` key to their custom observer schema, then the `metadata.namespace` field will be observed.

Observed fields

A field value will be observed if it is defined in the observer schema. Its value should be `null` (in YAML), except for fields used for the resource identification.

In the example above:

- the `spec.type` of the Service is not observed, as it is not present in the custom observer schema. Its original value is specified in the manifest file, but Krake doesn't guarantee this value to remain.
- the `spec.selector.app` of the `Service` is observed as it is present in the custom observer schema. Krake guarantee that its original value will remain the same, by observing the value and reverting any changes which were not made through Krake.
- the `spec.sessionAffinity` of the `Service` is observed. As it is not present in the manifest, the Kubernetes API will initialize it. Once it has been initialized by Kubernetes, Krake guarantee that its value will not be modified outside of Krake.

Warning: A non-observed field cannot be updated by Krake. In order to update such a field, one also need to observe it (i.e. update the custom observer schema to add this field).

Note: Except for the fields used for identifying the Kubernetes resource, all fields value MUST be `null`. Otherwise, the custom observer schema is invalid.

List length control

A list's length is controlled though the used of a special control dictionary, added as the last element of a list. The minimum and maximum length of the list must be specified.

In the example `Service`'s custom observer schema, the number of `ports` must be between 1 and 4. If the length of the `ports` list is below 1 or above 4, Krake reverts the `Service` to its last known value.

For the first port, the value of `port`, `protocol`, `targetPort` are defined in the manifest file.

The presence of a second element in the `ports` list in the custom observer schema doesn't guarantee its presence. Krake guarantee that, if a second port is set, its value won't be allowed to change outside of Krake. It can be removed and re-added, as long as its value remains unchanged.

Tip: Krake doesn't allow to set a minimum list length value below the number of element specified in the manifest file.

Tip: An unlimited list length can be specified by setting `observer_schema_list_max_length` to 0.

Note: A list MUST contain the special control dictionary. Otherwise, the custom observer schema is invalid.

2.3.3 Usage

A custom observer schema can be specified in `rok` with the argument `-O` or `--observer_schema`. If none is provided, a default observer schema is generated and all fields defined in `spec.manifest` are observed

2.4 User Stories

2.4.1 Introduction

This guide

This guide aims at providing an introduction in some concepts and mechanisms of Krake. It provides guidances and commands that readers are encouraged to try out by themselves on a demo environment as described in the next section.

It does not aim at providing an exhaustive list of commands nor all the possible ways how to use them.

This guide is structured into independent Scenarios which usually start with a Preparation section, and end with a Cleanup section.

Demo Environment

Note: The demo environment described in this section refers to a standard development environment deployed with Ansible. See *Set up Krake with Ansible*

The demo environment is comprised of 3 virtual machines in the same private network:

- The Krake VM: It runs all the Krake components in docker containers, as well as a Prometheus Server to simulate scheduling data for the backends.
- The two Minikube VMs `minikube-cluster-1` and `minikube-cluster-2`: They run an all-in-one Kubernetes "cluster". They are used as backends by Krake to deploy the users' applications.

Note: Scenario *OpenStack backends* additionally requires to have an OpenStack project at hand.

On the Krake VM, the two Kubernetes clusters `kubeconfig` files are present:

```
$ ll clusters/config/
$ cat clusters/config/minikube-cluster-1
$ cat clusters/config/minikube-cluster-2
```

Note: Unless stated otherwise (generally in the prompt), all commands are run on the Krake VM, with the `krake` user.

A simple manifest file will be used as a demo application. It can be found at the following path:

```
$ cat git/krake/rak/functionals/echo-demo.yaml
```

2.4.2 Demonstration of basic commands and workflow

Goal: Get familiar with basic `rok` commands, and with the associated internal Krake mechanisms.

Introduction to the `rok` CLI

- Following commands provide basic help on the `rok` CLI and its structure:

```
$ rok --help
$ rok kubernetes --help  # Similar to "rok kube --help"
$ rok kube application --help  # Similar to "rok app --help"
$ rok kube cluster --help
$ rok infrastructure --help  # Similar to "rok infra --help"
```

Register a cluster

- Register a Kubernetes cluster using its associated Kubernetes `kubeconfig` file.

```
$ rok kube cluster list  # No Cluster resource is present
$ rok kube cluster register -k clusters/config/minikube-cluster-1
$ rok kube cluster list  # One Cluster resource with name "minikube-cluster-1"
```

Note: The command `register` registers an existing Kubernetes cluster through its kubeconfig file. Resource called a `Cluster` (handled by the `kubernetes` API of Krake) is created by the `register` command. It contains multiple pieces of information, in particular the content of the kubeconfig file itself. The resource helps to store the information needed to connect to the actual Kubernetes cluster.

Important: In the following, a **Kubernetes cluster** refers to an actual cluster, which has been already installed and prepared. This can be the Minikube clusters deployed by the Krake test environment.

A **Krake Kubernetes Cluster** is a resource in the Krake database, which was created by Krake or registered into Krake and contains the kubeconfig file of the corresponding Kubernetes cluster.

Tip: Krake is able to actually **create** a Kubernetes cluster by supported infrastructure providers. If you are interested in the topic of Kubernetes cluster life-cycle management by Krake please refer to the *Infrastructure providers* section.

Spawn the demo application

- Spawn a Kubernetes `Application` using its Kubernetes manifest file.

```
$ rok kube app list  # No Application resource is present
$ rok kube app create -f git/krake/rak/functionals/echo-demo.yaml echo-demo
$ rok kube app list  # One Application resource with name "echo-demo"
```

– **Alternatively**, spawn a Kubernetes `Application` using a `TOSCA` template file (or URL) or `CSAR` archive URL, see *Examples*.

```
$ rok kube app list  # No Application resource is present
$ rok kube app create -f git/krake/rak/functionals/echo-demo-tosca.yaml echo-demo
$ rok kube app list  # One Application resource with name "echo-demo"
```

- Check application information:
  - Application Status is `RUNNING`.
  - Application is running on `minikube-cluster-1`.

```
$ rok kube app get echo-demo
$ rok kube app get echo-demo -o json  # Use JSON format, which is also more verbose
```

- Access the demo application endpoint:

```
$ APP_URL=$(rok kube app get echo-demo -o json | jq '.status.services["echo-demo"]'); APP_URL="$APP_U
$ curl $APP_URL
```

- Check the created resources on the Kubernetes cluster:

```
$ kubectl --kubeconfig clusters/config/minikube-cluster-1 get deployments
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
echo-demo   1/1     1            1           3h34m
$ kubectl --kubeconfig clusters/config/minikube-cluster-1 get services
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
echo-demo    NodePort    10.98.78.74   <none>        8080:32235/TCP   3h34m
kubernetes   ClusterIP   10.96.0.1     <none>        443/TCP          27h
$ kubectl --kubeconfig clusters/config/minikube-cluster-1 get po
NAME                         READY   STATUS    RESTARTS   AGE
echo-demo-6dc5d84869-4hcd8   1/1     Running   0          3h34m
```

Update resources

- Update the manifest file to create a second Pod for the `echo-demo` application.

```
$ cat git/krake/rak/functionals/echo-demo-update.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echo-demo
```

```
spec:
  replicas: 2
  selector:
    matchLabels:
      app: echo
  template:
    metadata:
      labels:
        app: echo
    spec:
      containers:
      - name: echo
        image: k8s.gcr.io/echoserver:1.9
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: echo-demo
spec:
  type: NodePort
  selector:
    app: echo
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
```

```
$ rok kube app update -f git/krake/rak/functionals/echo-demo-update.yaml echo-demo
```

– **Alternatively**, update a `TOSCA` template file (or URL) or `CSAR` archive URL to create a second Pod for the `echo-demo` application, see *Examples*.

```
$ rok kube app update -f git/krake/rak/functionals/echo-demo-update-tosca.yaml echo-demo
```

- Check the existing resources on the Kubernetes cluster: A second Pod has been spawned.

```
$ kubectl --kubeconfig clusters/config/minikube-cluster-1 get deployments
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
echo-demo   2/2     2            2           42m
$ kubectl --kubeconfig clusters/config/minikube-cluster-1 get po
NAME                         READY   STATUS        RESTARTS   AGE
echo-demo-6dc5d84869-2v6jh   1/1     Running       0          7s
echo-demo-6dc5d84869-l7fm2   1/1     Running       0          42m
```

Delete resources

- Issue the following commands to delete the `echo-demo` Kubernetes `Application` and the `minikube-cluster-1` Kubernetes `Cluster`.

```
$ rok kube app delete echo-demo
$ rok kube app list  # No Application resource is present
$ rok kube cluster delete minikube-cluster-1
$ rok kube cluster list  # No Cluster resource is present
```

2.4.3 Scheduling an Application using `Labels` and `LabelConstraints`

Goal: Explore the labels mechanisms and schedule an application based on labels and label constraints

Introduction to Scheduling mechanisms

Note: After its initial scheduling, an application location is re-evaluated every 60 seconds by the Scheduler - the so-called "rescheduling". In the following scenarios, we observe both the initial scheduling of an application, and a migration triggered by the rescheduling. To correctly observe this mechanism, it is recommended to check the Scheduler logs, the Application status, and the resources running on the Kubernetes clusters directly, by running the following commands in separate terminals:

- Watch the `echo-demo` Kubernetes `Application` status, and more precisely its current location:

```
$ watch "rok kube app get echo-demo -o json | jq .status.running_on"
```

- Watch Scheduler logs:

```
$ docker logs -f krake_krake-ctrl-scheduler_1
```

- Observe k8s resources on both Minikube clusters:

```
$ watch kubectl --kubeconfig clusters/config/minikube-cluster-1 get all
```

```
$ watch kubectl --kubeconfig clusters/config/minikube-cluster-2 get all
```

Preparation

- Register the two clusters with a `location` `Label`.

Note: Each label always have a key and a value. We follow the same specifications as Kubernetes.

```
$ rok kube cluster register -k clusters/config/minikube-cluster-1 -l location=DE
$ rok kube cluster register -k clusters/config/minikube-cluster-2 -l location=SK
```

Spawn the demo application

- Create an application with a `location` `LabelConstraints`, and observe where it is deployed.

```
$ rok kube app create -f git/krake/rak/functionals/echo-demo.yaml echo-demo -L location=DE
$ rok kube app get echo-demo -o json | jq .status.running_on
```

Observe a migration

- Update an application's `LabelConstraints` and observe the migration to the second Kubernetes cluster.

```
$ rok kube app update echo-demo -L location=SK
$ rok kube app get echo-demo -o json | jq .status.running_on  # The Application is now running on "m
```

Cleanup

- Delete the `echo-demo` Kubernetes `Application` and both Krake Kubernetes `Clusters`

```
$ rok kube app delete echo-demo
$ rok kube cluster delete minikube-cluster-1
$ rok kube cluster delete minikube-cluster-2
```

2.4.4 Scheduling an Application Using Metrics

Goal: Explore the metrics mechanisms and schedule an application based on cluster metrics.

Note: Refer to the *Introduction to Scheduling mechanisms* for useful commands to observe the migration mechanism.

Introduction

Metrics in the Krake sense have two meanings. The first one is an actual value for some parameter, which can be measured or computed in the real world. For instance the current space available on a data center, its latency, the amount of green energy used by the data center could all be metrics. This value may be dynamic and change over time. A `GlobalMetric` is also a resource in Krake. It represents an actual metric, is stored in the database, and defines a few elements, such as the minimum and maximum values (Krake only considers numbers for the metrics).

The Krake scheduler can use these metrics to compute the score of a Krake Kubernetes `Cluster`. Each cluster is associated with a list of metrics and their respective weights for this cluster. This list is defined by the user who added the `Cluster` resource into Krake. A higher weight means that the metric has a higher influence in the score: a metrics with a low value, but a high weight may have more impact on the score than a metric with medium value but low weight. The `Cluster` metrics and the computed score is then used in the `Application` scheduling process.

For Krake to fetch the current value of a metric, a user needs to define where and how it can be requested. `GlobalMetricsProvider` resources can be created for this purpose. They have different types, to support different technologies. There is for example a support for Prometheus. The `GlobalMetricsProvider` resource will define the URL of the Prometheus instance and some other metadata, and afterwards, Krake's scheduler can automatically fetch the current value of the metrics for the score's computation.

The static providers are simple metric providers usually only set during tests. They allow a Krake resource to be associated with simple metrics, for which the value can be fetched easily, without having to set up a whole infrastructure.

The static providers thus give values for `GlobalMetric` resources. This value is only defined in the resource (stored in the database). Updating the `GlobalMetricsProvider` resource definition thus implies updating the value of the metrics.

An example of a static `GlobalMetricsProvider` resource is given in the following. It is used in the next steps of this guide. As explained, the value of the metrics it provides are directly set inside its definition:

```
api: core
kind: GlobalMetricsProvider
metadata:
  created: '2020-01-21T10:50:11.500376'
  deleted: null
  finalizers: []
  labels: {}
  modified: '2020-01-21T10:50:11.500376'
  name: static_provider
  namespace: null
```

(continues on next page)

(continued from previous page)

```
  owners: []
  uid: 26ef45e8-e5c8-44fe-8a7f-a3f40944c925
spec:
  static:
    metrics:
      electricity_cost_1: 0.9  # Set the value that will be provided for this metric
      green_energy_ratio_1: 0.1  # Set the value that will be provided for this metric
  type: static
```

To get additional information about the metrics and metrics providers, please read the documentation about them, see *Metrics and Metrics Providers*.

Preparation

- Add the `static_provider` metrics provider using the bootstrap script (from the root of the Krake repository):

```
$ cd <path_to_krake_root>
$ krake_bootstrap_db support/static_metrics.yaml
```

- Check that the `GlobalMetricsProvider` and `GlobalMetrics` objects have been successfully added:

```
$ rok core globalmetricsprovider get static_provider
+-----------+--------------------------+
| name      | static_provider          |
| namespace | None                     |
| labels    | None                     |
| created   | 2000-01-01 08:00:00      |
| modified  | 2000-01-01 08:00:00      |
| deleted   | None                     |
| type      | static                   |
| metrics   | electricity_cost_1: 0.9  |
|           | green_energy_ratio_1: 0.1 |
+-----------+--------------------------+
$ rok core globalmetric get electricity_cost_1
+-----------+---------------------+
| name      | electricity_cost_1  |
| namespace | None                |
| labels    | None                |
| created   | 2000-01-01 08:00:00 |
| modified  | 2000-01-01 08:00:00 |
| deleted   | None                |
| provider  | static_provider     |
| min       | 0                   |
| max       | 1                   |
+-----------+---------------------+
$ rok core globalmetric get green_energy_ratio_1
+-----------+----------------------+
| name      | green_energy_ratio_1 |
| namespace | None                 |
| labels    | None                 |
| created   | 2000-01-01 08:00:00  |
| modified  | 2000-01-01 08:00:00  |
| deleted   | None                 |
| provider  | static_provider      |
| min       | 0                    |
```

```
| max        | 1                    |
+------------+----------------------+
```

- Register `minikube-cluster-1` and `minikube-cluster-2` clusters, and associate the `electricity_cost_1` and `green_energy_ratio_1` metrics to them using different weights to get different ranking scores:

```
$ rok kube cluster register -k clusters/config/minikube-cluster-1 --global-metric electricity_cost_1
$ rok kube cluster register -k clusters/config/minikube-cluster-2 --global-metric electricity_cost_1
```

- The clusters `minikube-cluster-1`/`-2` have been defined with the following weights for the two static metrics:

	`minikube-cluster-1`	`minikube-cluster-2`	Value
`electricity_cost_1`	Weight: 10	Weight: 1	0.9
`green_energy_ratio_1`	Weight: 1	Weight: 10	0.1
Score	**9.1**	**1.9**	

As the score of `minikube-cluster-1` is higher, it will been chosen, and the Application will be deployed on it. The score is computed like the following:

$$10 \cdot 0.9 + 1 \cdot 0.1 = 9.1$$

Scheduling of an application

- Create the `echo-demo` application and check it is actually deployed on the first cluster:

```
$ rok kube app create -f git/krake/rak/functionals/echo-demo.yaml echo-demo
$ rok kube app get echo-demo  # See "running_on": the Application is running on "minikube-cluster-1"
```

Note: You can observe the scheduler logs in `DEBUG` mode to gather additional understanding of the scheduling mechanism.

Observe a migration

- The Scheduler regularly performs a check, to ensure the current cluster on which an Application is running is the best, depending on its score. This check is done by default every minute (see the configuration of the *Scheduler*). If an available cluster with a better score than the one of the current cluster is found, the Application is migrated from the current to the better cluster.

  As the score is computed using the metrics, we can trigger the migration by updating the exported value of the metrics in the `static_provider` `GlobalMetricsProvider` resource. The following command updates the value of the static metrics:

  - `electricity_cost_1`: to have a value of 0.1;
  - `green_energy_ratio_1`: to have a value of 0.9;

	`minikube-cluster-1`	`minikube-cluster-2`	New value
`electricity_cost_1`	Weight: 10	Weight: 1	0.1
`green_energy_ratio_1`	Weight: 1	Weight: 10	0.9
Score	**1.9**	**9.1**	

Note: This is not the actual score but a simplification, as stickiness is also part of the computation, see *Scheduling of Applications*

- Update the value of the metrics, by updating the `static_provider` GlobalMetricsProvider:

```
$ rok core globalmetricsprovider update static_provider --metric electricity_cost_1 0.1 --metric gree
+-----------+--------------------------+
| name      | static_provider          |
| namespace | None                     |
| labels    | None                     |
| created   | 2021-04-08 08:04:23      |
| modified  | 2021-04-08 08:10:34      |
| deleted   | None                     |
| type      | static                   |
| metrics   | electricity_cost_1: 0.1  |
|           | green_energy_ratio_1: 0.9 |
+-----------+--------------------------+
```

- Now, by waiting a bit (maximum 60 seconds if you kept the default configuration), the Scheduler should have checked the new values of the metrics, and have requested a migration of the Application onto `minikube-cluster-2`, which has now the better score:

```
$ rok kube app get echo-demo  # See "running_on": the Application is running on "minikube-cluster-2"
```

Cleanup

- Delete the `echo-demo` Kubernetes `Application` and both Kubernetes `Clusters`.

```
$ rok kube app delete echo-demo
$ rok kube cluster delete minikube-cluster-1
$ rok kube cluster delete minikube-cluster-2
```

2.4.5 OpenStack backends

Warning: Due to stability and development issues on the side of Magnum, this feature isn't actively developed anymore.

Goal: Demonstrate the use of an OpenStack project as a backend for Krake

Note: Krake supports different kind of backends. In the previous example, we used a Kubernetes cluster (deployed in a single VM via Minikube). In this scenario, we register an existing OpenStack project.

Register an existing OpenStack project to Krake

- Gather information about your OpenStack project, for example:

```
ubuntu@myworkstation:~$ openstack coe cluster template list # Get Template ID
ubuntu@myworkstation:~$ openstack project show my_openstack_project # Get Project ID
ubuntu@myworkstation:~$ openstack user show my_user # Get User ID
ubuntu@myworkstation:~$ grep OS_AUTH_URL ~/openrc # Get Keystone auth URL
```

- Create a OpenStack `Project` resource in Krake:

```
$ rok os project create --template 728f024e-8a88-4971-b79f-151da123f363 --project-id 5bc3bab620bd48b0
```

Create a `MagnumCluster`

```
$ rok os cluster list  # No Cluster resource is present
$ rok os cluster create mycluster
$ rok os cluster list  # One Cluster resource with name "mycluster"
```

Note: The creation of the Magnum cluster can take up to 10 minutes to complete.

- Observe that one Kubernetes `Cluster` is created in association to the `MagnumCluster`.

```
$ rok kube cluster list
```

Spawn the demo application

- Create the demo Kubernetes `Application` and observe the resource status.

```
$ rok kube app create -f git/krake/rak/functionals/echo-demo.yaml echo-demo
$ rok kube app get echo-demo  # See "running_on"
```

Cleanup

- Delete the `echo-demo` Kubernetes `Application` and the OpenStack `Project`

```
$ rok kube app delete echo-demo
$ rok kube os cluster delete mycluster
$ rok kube project delete myproject
```

2.4.6 Creation and deployment of a stateful application

Goal: Create and deploy a stateful application to Krake.

Note: This feature is still under development in Krake, so new features could be added or removed in the future. Also, some implementation details might change.

Therefore, this page is subject to changes until this note is removed.

2.4.7 Infrastructure providers

Goal: Demonstrate the use of an OpenStack based cloud backend for Krake. Krake uses the IM (Infrastructure Manager) provider as a backend for spawning a Kubernetes cluster. A Kubernetes cluster is then automatically registered in Krake and could be used for the application deployment.

This is an advanced user scenario where the user should register the existing infrastructure provider backend (IM) as well as an existing IaaS cloud deployment (OpenStack) before the actual cluster creation. The user is navigated to register those resources to Krake. Please read the brief overview of Krake's infrastructure provider and cloud resources below:

Krake resources called **GlobalInfrastructureProvider** and **InfrastructureProvider** correspond to an infrastructure provider backend, that is able to deploy infrastructures (e.g. Virtual machines, Kubernetes clusters, etc.) on IaaS cloud deployments (e.g. OpenStack, AWS, etc.). Krake currently supports IM (Infrastructure Manager) as an infrastructure provider backend.

Krake resources called **GlobalCloud** and **Cloud** correspond to an IaaS cloud deployment (e.g. OpenStack, AWS, etc.) that will be managed by the infrastructure provider backend. GlobalCloud and Cloud resources could contain also metrics and labels, that could be used in Cluster scheduling. Krake currently supports OpenStack as a GlobalCloud or Cloud backend.

Note: The *global* resource (e.g. GlobalInfrastructureProvider, GlobalCloud) is a non-namespaced resource that could be used by any (even namespaced) Krake resource. For example, the GlobalCloud resource could be used by any Cluster which needs to be scheduled to some cloud.

Note: Note that file paths mentioned in this tutorial are relative to the root of the Krake repository.

Preparation

Launch the IM infrastructure provider instance using the support script. Please note that the IM instance is launched in the docker environment, therefore it is mandatory to install docker beforehand.

```
support/im
```

Warning: The above support script launches the IM as is described in the IM quick start tutorial, hence with the default configuration and in a non-productive way. Please visit the IM documentation for further information about how to configure, launch and interact with the IM software.

Register an existing infrastructure provider to Krake

IM service `username` and `password` could be arbitrary. The `username` and `password` are used for userspace definition. It means, that anyone can talk with IM but can see only their own userspace.

```
rok infra provider register --type im --url http://localhost:8800 --username test --password test im-
```

Register an existing OpenStack based cloud to Krake

Gather information about your OpenStack project from `openrc` file:

- Insert the `OS_AUTH_URL` value (without path) to the `--url` argument, e.g. *https://identity.cloud.com:5000*
- Insert the `OS_PROJECT_NAME` value to the `--project` argument
- Insert the `OS_USERNAME` value to the `--username` argument
- Insert the `OS_PASSWORD` value to the `--password` argument

Use the already registered infrastructure provider called `im-provider` as an infrastructure provider for your OpenStack cloud.

Note: If you want to use the C&H F1A OpenStack cloud, please note that it does not assign public IPs to the VMs. C&H F1A requires a private network for all VMs. This private network is connected to the public one via a router.

The router should be created beforehand as the IM provider is not able to do this. You can create the requested router in your C&H F1A OpenStack cloud project as follows:

```
openstack router create --external-gateway shared-public-IPv4 public_router
```

```
rok infra cloud register --type openstack --url <os-auth-url> --project <os-project-name> --username
```

Create a Cluster

```
rok kube cluster list  # No Cluster resource is present
rok kube cluster create -f rak/functionals/im-cluster.yaml my-cluster
rok kube cluster list  # One Cluster resource with name "my-cluster"
```

The creation of the cluster can take up to 15 minutes to complete. Observe that Kubernetes Cluster is created.

```
rok kube cluster list
```

Spawn the demo application

Create the demo Kubernetes Application and observe the resource status.

```
rok kube app create -f rak/functionals/echo-demo.yaml echo-demo
rok kube app get echo-demo  # See "running_on"
```

Cleanup

Delete the Cluster, the Cloud and the InfrastructureProvider:

```
rok kube cluster delete my-cluster
rok infra cloud delete os-cloud
rok infra provider delete im-provider
```

2.4.8 Scheduling a Cluster using `Labels` and `LabelConstraints`

Goal: Explore the labels mechanisms and schedule a Cluster based on labels and label constraints.

Introduction

Note: Refer to the *Label constraints* for useful information about label constraints.

Krake allows the user to define a label constraint and restrict the deployment of Cluster resources only to cloud backends that match **all** defined labels.

Preparation

Please go through the *Preparation* as well as through the *Register an existing infrastructure provider to Krake* and register an infrastructure provider. Validate the infrastructure provider registration as follows:

```
$ rok infra provider list
+-------------+--------------+--------+---------------------+---------------------+---------+------+-
|    name     |  namespace   | labels |       created       |      modified       | deleted | type |
+=============+==============+========+=====================+=====================+=========+======+=
| im-provider | system:admin | None   | 2000-01-01 08:00:00 | 2000-01-01 08:00:00 | None    | im   |
+-------------+--------------+--------+---------------------+---------------------+---------+------+-
```

Register `os-cloud-1` and `os-cloud-2` Clouds, and associate the *location* Label. Each label always has a key and a value. We follow the same specifications as the Kubernetes project.

Note: Refer to the *Register an existing OpenStack based cloud to Krake* for useful information about Cloud attributes.

```
rok infra cloud register -l location=DE --type openstack --url <os-auth-url> --project <os-project-na
rok infra cloud register -l location=SK --type openstack --url <os-auth-url> --project <os-project-na
```

Tip: You do not need access to the two OpenStack projects for `os-cloud-1` and `os-cloud-2` registration. It is possible to register one OpenStack project two times in Krake with different labels. Do not use this setup in the production environment!

Scheduling of a Cluster

Create `my-cluster` cluster with a *location* LabelConstraints, and observe where it is spawned.

```
rok kube cluster create -f git/krake/rak/functionals/im-cluster.yaml my-cluster -L location=SK
rok kube cluster get my-cluster -o json | jq .status.running_on  # Cluster is running on "os-cloud-2"
```

Note: You can observe the scheduler logs in *DEBUG* mode to gather additional understanding of the scheduling mechanism.

Cleanup

Delete the Cluster, both Clouds and the IM InfrastructureProvider.

```
rok kube cluster delete my-cluster
rok infra cloud delete os-cloud-1
rok infra cloud delete os-cloud-2
rok infra provider delete im-provider
```

2.4.9 Scheduling a Cluster using Metrics

Goal: Schedule the cluster based on cloud metrics.

Introduction

Note: Refer to the *Introduction* and *Metrics and Metrics Providers* for useful information about metrics.

The Krake scheduler can use metrics to compute the score of a Krake Cloud resource. Each Cloud is associated with a list of metrics and their respective weights for this Cloud. This list is defined by the user who added the Cloud resource into Krake. A higher weight means that the metric has a higher influence in the score: a metric with a low value, but a high weight may has more impact on the score than a metric with a medium value but low weight. The Cloud metrics and the computed score are then used in the Cluster scheduling process.

Note: Note that file paths mentioned in this tutorial are relative to the root of the Krake repository.

Preparation

Add the `static_provider` metrics provider using the bootstrap script (from the root of the Krake repository):

```
cd <path_to_krake_root>
krake_bootstrap_db support/static_metrics.yaml
```

Check that the GlobalMetricsProvider and GlobalMetrics objects have been successfully added:

```
$ rok core globalmetricsprovider list
+-----------------+-----------+--------+---------------------+---------------------+---------+-------
|      name       | namespace | labels |       created       |      modified       | deleted | mp_typ
+=================+===========+========+=====================+=====================+=========+=======
| static_provider | None      | None   | 2000-01-01 08:00:00 | 2000-01-01 08:00:00 | None    | static
+-----------------+-----------+--------+---------------------+---------------------+---------+-------

$ rok core globalmetric list
+----------------------+-----------+--------+---------------------+---------------------+---------+--
|         name         | namespace | labels |       created       |      modified       | deleted |
+======================+===========+========+=====================+=====================+=========+==
| electricity_cost_1   | None      | None   | 2000-01-01 08:00:00 | 2000-01-01 08:00:00 | None    | s
| green_energy_ratio_1 | None      | None   | 2000-01-01 08:00:00 | 2000-01-01 08:00:00 | None    | s
+----------------------+-----------+--------+---------------------+---------------------+---------+--
```

Please go through the *Preparation* as well as through the *Register an existing infrastructure provider to Krake* and register an infrastructure provider. Validate the infrastructure provider registration as follows:

```
$ rok infra provider list
+-------------+--------------+--------+---------------------+---------------------+---------+------+-
|    name     |  namespace   | labels |       created       |      modified       | deleted | type |
+=============+==============+========+=====================+=====================+=========+======+=
| im-provider | system:admin | None   | 2000-01-01 08:00:00 | 2000-01-01 08:00:00 | None    | im   |
+-------------+--------------+--------+---------------------+---------------------+---------+------+-
```

Register `os-cloud-1` and `os-cloud-2` Clouds, and associate the `electricity_cost_1` and `green_energy_ratio_1` metrics to them using different weights to get different ranking scores:

Note: Refer to the *Register an existing OpenStack based cloud to Krake* for useful information about Cloud attributes.

```
rok infra cloud register --global-metric electricity_cost_1 1 --global-metric green_energy_ratio_1 10
rok infra cloud register --global-metric electricity_cost_1 10 --global-metric green_energy_ratio_1 1
```

Tip: You do not need access to the two OpenStack projects for `os-cloud-1` and `os-cloud-2` registration. It is possible to register one OpenStack project two times in Krake with different metrics. Do not use this setup in the production environment!

The clouds `os-cloud-1/-2` have been defined with the following weights for the two static metrics:

	os-cloud-1	os-cloud-2	Value
electricity_cost_1	Weight: 1	Weight: 10	0.9
green_energy_ratio_1	Weight: 10	Weight: 1	0.1
Score	**1.9**	**9.1**	

As the score of `os-cloud-2` is higher, it will been chosen, and the Cluster will be spawned on it. The score is computed like the following:

$$10 \cdot 0.9 + 1 \cdot 0.1 = 9.1$$

Scheduling of a Cluster

Create the `my-cluster` cluster and check it is actually spawned on the second cloud:

```
rok kube cluster create -f rak/functionals/im-cluster.yaml my-cluster
rok kube cluster get my-cluster  -o json | jq .status.running_on  # Cluster is running on "os-cloud-2
```

Note: You can observe the scheduler logs in *DEBUG* mode to gather additional understanding of the scheduling mechanism.

Cleanup

Delete the Cluster, both Clouds and the InfrastructureProvider.

```
rok kube cluster delete my-cluster
rok infra cloud delete os-cloud-1
rok infra cloud delete os-cloud-2
rok infra provider delete im-provider
```

2.4.10 Horizontal Cluster Scaling

Goal: Scale up and then down (horizontally) the actual Kubernetes cluster using Krake.

This is an advanced user scenario where the user should register an existing infrastructure provider backend (IM) as well as an existing IaaS cloud deployment (OpenStack) before the actual cluster creation and scaling it horizontally. Horizontal scaling is the act of adding (or removing) nodes of the same size to the cluster.

Note: Keep in mind that Krake is able to actually **create** and then **scale** (update) the Kubernetes cluster by supported infrastructure providers. Please refer to the *Infrastructure Controller* and visit related user stories for more information about how the actual Kubernetes cluster could be managed by Krake.

Note: Note that file paths mentioned in this tutorial are relative to the root of the Krake repository.

Preparation

Please go through the *Preparation* as well as through the *Register an existing infrastructure provider to Krake* and register an infrastructure provider. Validate the infrastructure provider registration as follows:

```
$ rok infra provider list
+-------------+--------------+--------+---------------------+---------------------+---------+------+-
|    name     |  namespace   | labels |       created       |      modified       | deleted | type |
+=============+==============+========+=====================+=====================+=========+======+=
| im-provider | system:admin | None   | 2000-01-01 08:00:00 | 2000-01-01 08:00:00 | None    | im   |
+-------------+--------------+--------+---------------------+---------------------+---------+------+-
```

Please go through the *Register an existing OpenStack based cloud to Krake* and register an existing OpenStack cloud to Krake. Validate the cloud registration as follows:

```
$ rok infra cloud list
+----------+--------------+--------+---------------------+---------------------+---------+-----------
|   name   |  namespace   | labels |       created       |      modified       | deleted |   type
+==========+==============+========+=====================+=====================+=========+===========
| os-cloud | system:admin | None   | 2000-01-01 08:00:00 | 2000-01-01 08:00:00 | None    | openstack
+----------+--------------+--------+---------------------+---------------------+---------+-----------
```

Create the Cluster

Create the `my-cluster` cluster using the example TOSCA template stored in *rak/functionals/im-cluster.yaml*. This TOSCA template should create a Kubernetes cluster with one control plane node and one worker node.

```
rok kube cluster create -f rak/functionals/im-cluster.yaml my-cluster
```

The creation of the cluster can take up to 15 minutes to complete. The fully created and configured cluster should be in the *ONLINE* state. You should also see that 2 from 2 nodes total are healthy (nodes: 2/2). Validate them as follows:

```
$ rok kube cluster get my-cluster
+----------------------+-------------------------------------------------------------------------------
| name                 | my-cluster
| namespace            | system:admin
| labels               | None
| created              | 2000-01-01 08:00:00
| modified             | 2000-01-01 08:00:00
| deleted              | None
| state                | ONLINE
| reason               | None
| custom_resources     | []
| metrics              | []
| failing_metrics      | None
| label constraints    | []
| metric constraints   | []
| scheduled_to         | 'namespace': 'system:admin', 'kind': 'Cloud', 'name': 'os-cloud', 'api': '
| scheduled            | 2000-01-01 08:00:00
| running_on           | 'namespace': 'system:admin', 'kind': 'Cloud', 'name': 'os-cloud', 'api': '
| nodes                | 2/2
| nodes_pid_pressure   | 0/2
| nodes_memory_pressure | 0/2
| nodes_disk_pressure  | 0/2
+----------------------+-------------------------------------------------------------------------------
```

Optionally, you can export the `my-cluster` kubeconfig file and validate the cluster health and nodes count directly by the kubectl CLI. You can do this as follows (with the help of jq command-line JSON processor):

```
rok kube cluster get my-cluster -o json | jq .spec.kubeconfig > kubeconfig.json
```

Access the `my-cluster` cluster:

```
$ kubectl --kubeconfig=kubeconfig.json get nodes
NAME                      STATUS   ROLES                  AGE     VERSION
kubeserver.localdomain    Ready    control-plane,master   10m     v1.22.9
vnode-1.localdomain       Ready    <none>                 9m46s   v1.22.9
```

Scale up the Cluster

Scale the created cluster up using the example TOSCA template stored in *rak/functionals/im-cluster-scale-up.yaml*. This TOSCA template should add one worker node. Its size (flavor) should be the same as the size of the previously created worker node.

> Alternatively, you can adjust the worker node number on your own. In this case, find and adjust the `wn_num` variable count in the TOSCA template:
>
> ```
> wn_num:
>   type: integer
>   description: Number of WNs in the cluster
>   default: 2
>   required: yes
> ```

Scale up the cluster:

```
rok kube cluster update -f rak/functionals/im-cluster-scale-up.yaml my-cluster
```

The scaling of the cluster can take up to 5 minutes to complete. The fully scaled and configured cluster should be in the *ONLINE* state. You should also see that one node has been successfully added i.e. 3 from 3 nodes total are healthy (nodes: 3/3). Validate them as follows:

```
$ rok kube cluster get my-cluster
+----------------------+-------------------------------------------------------------------------------------
| name                 | my-cluster
| namespace            | system:admin
| labels               | None
| created              | 2000-01-01 08:00:00
| modified             | 2000-01-01 08:00:00
| deleted              | None
| state                | ONLINE
| reason               | None
| custom_resources     | []
| metrics              | []
| failing_metrics      | None
| label constraints    | []
| metric constraints   | []
| scheduled_to         | 'namespace': 'system:admin', 'kind': 'Cloud', 'name': 'os-cloud', 'api': '
| scheduled            | 2000-01-01 08:00:00
| running_on           | 'namespace': 'system:admin', 'kind': 'Cloud', 'name': 'os-cloud', 'api': '
| nodes                | 3/3
| nodes_pid_pressure   | 0/3
| nodes_memory_pressure | 0/3
| nodes_disk_pressure  | 0/3
+----------------------+-------------------------------------------------------------------------------------
```

Access the `my-cluster` cluster again and validate the cluster health and nodes count directly by the kubectl CLI:

```
$ kubectl --kubeconfig=kubeconfig.json get nodes
NAME                      STATUS     ROLES                  AGE    VERSION
kubeserver.localdomain    Ready      control-plane,master   34m    v1.22.9
```

```
vnode-1.localdomain      Ready      <none>                 32m    v1.22.9
vnode-2.localdomain      NotReady   <none>                 9m8s   v1.22.9
```

Scale down the Cluster

Scale the created cluster down using the example TOSCA template stored in *rak/functionals/im-cluster-scale-down.yaml*. This TOSCA template should remove one worker node.

Alternatively, you can adjust the worker node number on your own. In this case, find and adjust the `wn_num` and `removal_list` variables in the TOSCA template:

```
wn_num:
  type: integer
  description: Number of WNs in the cluster
  default: 1
  required: yes

...

wn:
  type: tosca.nodes.indigo.Compute
  capabilities:
    scalable:
      properties:
        count: { get_input: wn_num }
        removal_list: ['2']
```

The `removal_list` variable should be defined and should contain the ID(s) of the VM(s) which should be removed from the cluster. You can find the VM IDs in the `cluster.status.nodes` section of the Krake cluster resource as follows (with the help of jq command-line JSON processor):

```
$ rok kube cluster get my-cluster -o json | jq .status.nodes[].metadata.name
"kubeserver.localdomain"
"vnode-1.localdomain"
"vnode-2.localdomain"
```

Find the more detailed description about `removal_list` in the IM documentation.

Scale down the cluster:

```
rok kube cluster update -f rak/functionals/im-cluster-scale-down.yaml my-cluster
```

The scaling of the cluster can take up to 5 minutes to complete. The fully scaled and configured cluster should be in the *ONLINE* state. You should also see that one node has been successfully removed i.e. 2 from 2 nodes total are healthy (nodes: 2/2). Validate them as follows:

```
$ rok kube cluster get my-cluster
+----------------------+-------------------------------------------------------------------------------
| name                 | my-cluster
| namespace            | system:admin
| labels               | None
| created              | 2000-01-01 08:00:00
| modified             | 2000-01-01 08:00:00
| deleted              | None
| state                | ONLINE
| reason               | None
| custom_resources     | []
| metrics              | []
| failing_metrics      | None
| label constraints    | []
```

```
| metric constraints    | []
| scheduled_to          | 'namespace': 'system:admin', 'kind': 'Cloud', 'name': 'os-cloud', 'api': '
| scheduled             | 2000-01-01 08:00:00
| running_on            | 'namespace': 'system:admin', 'kind': 'Cloud', 'name': 'os-cloud', 'api': '
| nodes                 | 2/2
| nodes_pid_pressure    | 0/2
| nodes_memory_pressure | 0/2
| nodes_disk_pressure   | 0/2
+-----------------------+-------------------------------------------------------------------------
```

Access the `my-cluster` cluster again and validate the cluster health and nodes count directly by the kubectl CLI:

```
$ kubectl --kubeconfig=kubeconfig.json get nodes
NAME                     STATUS   ROLES                  AGE   VERSION
kubeserver.localdomain   Ready    control-plane,master   40m   v1.22.9
vnode-1.localdomain      Ready    <none>                 38m   v1.22.9
```

Cleanup

Delete the Cluster, Cloud and the InfrastructureProvider.

```
rok kube cluster delete my-cluster
rok infra cloud delete os-cloud
rok infra provider delete im-provider
```

2.5 HTTP Problem documentation

The failure reason of the Krake API HTTP layer is stored as an RFC7807 Problem. It is a way to define uniform, machine-readable details of errors in an HTTP response.

In case of a failure on the Krake API HTTP layer, the Krake API responds with a well-formatted RFC7807 Problem message, which could contain the following fields:

`type` A URI reference that identifies the problem type. It should point the Krake API users to the concrete part of the Krake documentation where the problem type is explained in detail. Defaults to about:blank.

`title` A short, human-readable summary of the problem type

`status` The HTTP status code

`detail` A human-readable explanation of the problem

`instance` A URI reference that identifies the specific occurrence of the problem

When RFC7807 Problem `type` is defined, it points Krake API clients to the below list of better-described HTTP problems.

Note: The RFC7807 Problem `type` URI is generated based on `--docs-problem-base-url` API configuration parameter (see *Krake configuration*) and the value from the list of defined HTTP Problems, see *`krake.api.helpers.HttpProblemTitle`*.

2.5.1 not-found-error

A requested resource cannot be found in the database.

2.5.2 transaction-error

A database transaction failed.

2.5.3 update-error

A update of resource field.

2.5.4 invalid-keystone-token

An authentication attempt with a keystone token failed.

2.5.5 invalid-keycloak-token

An authentication attempt with a keycloak token failed.

2.5.6 resource-already-exists

A resource already defined in the database is requested to be created.

CHAPTER 3

Administrator Documentation

This chapter presents the installation of Krake using Ansible.

3.1 Set up Krake with Ansible

This sections describes prerequisites and deployment of a Krake infrastructure with Ansible.

3.1.1 Prerequisites

- Ansible 2.9.x or superior using a Python 3 interpreter
- The full Openstack client python module
- Docker Python module

It is suggested that Ansible is installed inside the virtualenv of Krake.

```
pip install "ansible>=2.9" docker openstackclient
```

Check the version and Python executable of the Ansible installation:

```
ansible --version
```

```
ansible 2.9.2
  config file = None
  configured module search path = ['/path/to/home/.ansible/plugins/modules', '/usr/
↪share/ansible/plugins/modules']
  ansible python module location = /path/to/virtualenv/lib/python3.6/site-packages/
↪ansible
  executable location = /path/to/virtualenv/bin/ansible
  python version = 3.6.8 (default, Oct  7 2019, 12:59:55) [GCC 8.3.0]
```

3.1.2 Krake infrastructure deployment

The Krake infrastructure is provisioned by a set of Ansible playbooks. The infrastructure is split into multiple separate OpenStack Heat stacks. Every Heat stack is provisioned by its own Ansible playbook. The complete infrastructure can be created by the top-level **site.yml** playbook.

Krake YAML inventory file **hosts.yml** needs to be created. Use the example file and adjust it. The only parameter that needs to be modified is the **keypair** variable. It should name an existing OpenStack **keypair**. If the corresponding key file is not **~/.ssh/id_rsa** specify it in the **key_file** parameter.

It is assumed that environmental variables for authentication against OpenStack project exist. They can be set by sourcing the OpenStack RC file.

```
cd ansible/
cp hosts.yml.example hosts.yml
# Get list of all existing OpenStack keypairs
openstack keypair list
```

The complete infrastructure can be created by the top-level **site.yml** playbook.

```
ansible-playbook -i hosts.yml site.yml
```

Each infrastructure component can by created separately by corresponding ansible playbook e.g. the Krake application infrastructure can be created by Krake playbook.

```
ansible-playbook -i hosts.yml krake.yml
```

Table of available playbooks:

Krake Infrastructure component	Playbook name
Top-level playbook	site.yml
Krake application	krake.yml
Central IdP instance	central_idp.yml
Devstack instance	devstack.yml
Gateway SSH jump host	gateway.yml
Network “virtual” host	network.yml
Prometheus server instance	prometheus.yml
Minikube cluster	minikube_cluster.yml
Magnum cluster	magnum_cluster.yml

3.1.3 Krake Ansible directory structure

Ansible related-files are stored in the **ansible/** directory of the repository. Each sub-directory groups files based on Ansible best practices recommendations.

Sub-directory / File	Description
ansible.cfg	Local Ansible configuration
hosts.yml	Krake YAML inventory file
plugins/	Custom Ansible plugins
files/	Heat stack templates and files
plugins/	Custom Ansible plugins
group_vars/	Ansible group variables used as default values
roles/	Files for reusable Ansible roles
utils/	Krake Ansible helper scripts

3.1.4 Access through the gateway

To compartmentalize the infrastructure, all machines deployed by Krake are present on the same OpenStack private network. Only the gateway is associated with a floating IP and can thus be accessed externally. All other machines can be reached through the gateway.

To simplify this process, the wireguard VPN is installed on the gateway when deployed. After the deployment, for each wireguard peer set for the gateway in the host file (see *Inventory structure*), a wireguard configuration file is created in the etc directory where the inventory files are created (`ansible/.etc` by default). The files names have the following syntax: `wg_<peer_name>.conf`.

To use this file you have to:

0. install wireguard locally. If you are using Ubuntu, you can use the following command:

```
$ sudo apt install wireguard
```

1. generate a wireguard key:

```
$ umask 077
$ wg genkey > privatekey
$ wg pubkey < privatekey > publickey
```

2.1 open the `wg_<peer_name>.conf` and change the `REPLACEME` placeholder with the private key that corresponds to the peer.

2.2 Use SSH to connect to the `krake-gateway-server`. Check the gateway server and if necessary adjust to accommodate the correct wireguard keys. Replace the `REPLACEME` placeholder with the public key. You can find the public key in the directory under `/etc/wireguard`:

```
 [Interface]
 PrivateKey = <INSERT_PRIVATE_WIREGUARD_KEY>
 Address = 10.9.0.1

 [Peer]
 PublicKey = <INSERT_PUBLIC_KEY_FROM_GATEWAY_SERVER>

Endpoint = 185.128.119.165:51820
AllowedIPs = 10.9.0.0/24, 192.168.0.0/24
```

3. bring the wireguard interface up by using:

```
$ wg-quick up <path_to_file>/wg_<peer_name>.conf

# Example:
$ wg-quick up ansible/.etc/wg_my-peer.conf
```

4. you can now SSH into the other machines on the private network:

```
$ ssh ubuntu@<krake_VM_private_ip>
```

The wireguard interface can be brought down by using:

```
$ wg-quick down <path_to_file>/wg_<peer_name>.conf

# Example:
$ wg-quick down ansible/.etc/wg_my-peer.conf
```

Important: If several Krake deployments are managed from a single machine, the peer names should have a different value, to avoid conflicts with the wireguard network interfaces.

If several network interfaces are up at the same time, then the Krake private networks should not overlap. So if one has for instance the CIDR `192.168.0.0/24`, another deployment should use something independent, such as `192.168.1.0/24`.

3.2 Variables

This sections describes Krake Ansible variables definition. Variables are stored in the **group_vars/** directory or directly in the inventory file **hosts.yml**. Inventory file defines variables on global level or on a host group level (see *Inventory*).

3.2.1 Variables definition

Krake Ansible variables stored in **group_vars/** directory are structured into files where the filename matches the inventory host group name. Global variables common for all inventory host groups are defined in **all.yml** variable file. Following section describes files and variables used by Krake Ansible playbooks.

all.yml

etc_dir The directory of JSON files which store inferred information from hosts by **krake_inventory** plugin

central_idps.yml

keystone_port Central IdP keystone port

secgroup_name Security group name

flavor Flavor manages the sizing for the compute, memory and storage capacity of the host

floating_ip Enables the use of public IP address

git_branch Git branch name

devstack.yml

prometheus_port Prometheus server port

service_provider_port Prometheus clients port

template_name Name of the default kubernetes cluster template

cluster_keypair Name of the default keypair that is used to spawn kubernetes clusters via Magnum

idp_name Central IdP name

federated_domain_name Federated domain name

federation_protocol_name Federation protocol name

idp_mapping_name Central IdP mapping name

flavor Flavor manages the sizing for the compute, memory and storage capacity of the host

floating_ip Enables the use of public IP address

git_branch Git branch name

gateways.yml

flavor Flavor manage the sizing for the compute, memory and storage capacity of the host

wireguard_port: Port on which the wireguard service will listen to on the gateway.

krake_apps.yml

flavor Flavor manages the sizing for the compute, memory and storage capacity of the host

floating_ip Enables the use of public IP address

api_host OpenStack Heat template API name

api_port OpenStack Heat template API port

etcd_host (string) Name of the etcd container started using docker-compose.

etcd_port (int) Port of the etcd cluster in the etcd container started using docker-compose.

etcd_peer_port (int) Peer listening port of the etcd cluster in the etcd container started using docker-compose.

api_host (string) Name of the Krake API container started using docker-compose.

api_port (int) Port that can be used to reach the Krake API present in the container started using docker-compose.

enable_tls (boolean) Enable or disable TLS support for communications with the API (for the API, controllers and Rok utility). The certificates need to be added manually into the `/etc/krake` directory in the Krake VM.

worker_count (integer) On each Controller, amount of working units that will handle resources received concurrently.

debounce (float) On each Controller, timeout (in seconds) for the worker queue before handing over a resource, to wait for an updated state.

complete_hook_user (string) Name of the user for the "complete" hook.

complete_hook_cert_dest (file path) Path inside the deployed Application where the certificate and its key will be stored (for the "complete" hook).

complete_hook_env_token (string) Name of the environment variable that will contain the token in the deployed Application.

complete_hook_env_url (string) Name of the environment variable that will contain the URL of the Krake API in the deployed Application.

external_endpoint (URL, optional) URL of the Krake API that will be reachable for any deployed Application.

use_private_ip (boolean) If set to True, and no external endpoint has been set, the URL for the external endpoint (see above) will be computed automatically, using the Krake API private IP, its port and the "http" or "https" scheme depending on the status of TLS on the Krake API (enabled or disabled).

shutdown_hook_user (string) Name of the user for the "shutdown" hook.

shutdown_hook_cert_dest (file path) Path inside the deployed Application where the certificate and its key will be stored (for the "shutdown" hook).

shutdown_hook_env_token (string) Name of the environment variable that will contain the token in the deployed Application.

shutdown_hook_env_url (string) Name of the environment variable that will contain the URL of the Krake API in the deployed Application.

magnum_clusters.yml

prometheus_port Prometheus server port

magnum_path Magnum path

kube_api_config Path of kubernetes configuration file

user_role Federated user role

user_project Federated project name

minikube_clusters.yml

api_port OpenStack Heat template api port

minikube_install_dir Minikube installation directory path

minikube_version Minikube version

kubectl_version Kubectl version

kube_api_config Kubectl api configuration file path

minikube_path Minikube keystone path

user_role Federated user role

user_project Federated project name

flavor Flavor manages the sizing for the compute, memory and storage capacity of the host

floating_ip Enables the use of public IP address

prometheus.yml

prometheus_admin_pass Prometheus server admin password

grafana_admin_pass Grafana server admin password

ports Prometheus server VM open ports

flavor Flavor manages the sizing for the compute, memory and storage capacity of the host

floating_ip Enables the use of public IP address

git_branch Git branch name

3.3 Inventory

This sections describes the Ansible inventory of the Krake project. Ansible works against multiple infrastructure hosts. Hosts are configured in an inventory file **hosts.yml** which is a standard Ansible YAML inventory that uses multiple groups structure and a custom **krake_inventory** plugin (see auto plugin).

3.3.1 Inventory plugin

The **krake_inventory** custom plugin loads the JSON file defined by variable **hosts_file** and augments the host variables with dynamic variables (e.g. public and private IP addresses) depending on host. The location of JSON file which stores inferred information can be configured by specifying the **hosts_file** variable in the **all** group . If it is not specified it defaults to **.etc/<inventory-filestem>.json**.

3.3.2 Inventory structure

Krake inventory file **hosts.yml** uses Ansible multiple groups structure of inventory.

Global variables for all hosts are defined under the `vars` sub-section. This sub-section defines following:

keypair OpenStack SSH key pair name of public ssh key which will be used for accessing the infrastructure to deploy hosts. Different keys could be defined directly for specific group or host.

key_file SSH private key file path on local computer for corresponding **keypair**. If **key_file** is set to null, the default SSH identity (~/.ssh/id_rsa) will be used.

gateway SSH jump host that is used to access the OpenStack instances. By default, no OpenStack server has a floating IP assigned except hosts in the **gateways** group. All other hosts use the **gateway** host variable to define a SSH jump host. Wireguard is also installed on the gateway, see *Access through the gateway*

authorized_keys - optional List of additional authorized SSH keys, which can be used for accessing the hosts.

Each Krake infrastructure host is defined by corresponding host group sub-section in Krake inventory file. The default parameters for every host group are defined in the **group_vars/** directory where the filename matches the group name. Krake inventory file defines following host groups and host variables:

gateways SSH jump host that is used to access the OpenStack instances.

network Inventory name of the network on which this SSH jump host should be deployed

vpn_cidr VPN Classless Inter-Domain Routing definition (e.g. 10.9.0.0/24). This will define the wireguard network. Each peer on this network (the gateway and users or administrators of the deployment) will have a specific address on this network.

wireguard_peers List of all wireguard peer for whom access should be granted on the gateway. Several peers can be added. A wireguard configuration file will be created for each peer.

name The name of the peer. This string is used to differentiate the different peers from each other. It will also be given to the wireguard network interface. The value can be arbitrary, but should be unique per deployment, or over deployment if you plan on managing several ones with the same machine.

public_key The wireguard public key of the peer.

IP Set the IP that will be given to the current peer in the wireguard network. Each peer should be given a different IP to prevent conflicts. The IP can be chosen in the `vpn_cidr` network, as long as it is not the IP given to the gateway (which is the first in the network by default).

networks Networks group define "virtual" hosts. These hosts exist purely for provisioning purpose. No machines are associated with them.

subnet_name Subnet name

subnet_cidr Subnet Classless Inter-Domain Routing definition (e.g. 192.168.0.0/24)

public_network Public network type (e.g. shared-public-IPv4)

router_name Router name

common_secgroup_name Secure group name

central_idps Central IdP host group used for keystone federation of Krake infrastructure.

network Inventory name of the network on which this IdP should be deployed

devstacks Devstack host group used for deployment of Krake devstack backends.

id Unique DevStack ID. This ID is also used to define the IP network of the DevStack instance in the private network

network Inventory name of the network on which this DevStack should be deployed

idp Inventory name of the IdP that should be used for federation by this DevStack

prometheus Inventory name of the Prometheus server that should be used for the monitoring of this DevStack backend

magnum_clusters Magnum cluster host group used for deployment of magnum clusters on underlying devstack backend.

- **name** Magnum cluster name
- **devstack** Inventory name of underlying devstack backend which hosted the magnum cluster deployment
- **prometheus** Inventory name of the Prometheus server that should be used for the monitoring of this magnum cluster
- **use_keystone** Enables keystone deployment on this magnum cluster

minikube_clusters Minikube cluster host group used for deployment of minikube clusters.

- **name** Minikube cluster name
- **network** Inventory name of the network on which this minikube cluster should be deployed
- **idp** Inventory name of the IdP that should be used for federation by this minikube
- **use_keystone** Enables keystone deployment on this minikube cluster

prometheus Prometheus host group used for deployment of Prometheus monitoring server.

- **hostname** Prometheus VM host name
- **network** Inventory name of the network on which this minikube cluster should be deployed

krake_apps Krake application host group used for deployment Krake infrastructure

- **hostname** Krake VM host name
- **network** Inventory name of the network on which this minikube cluster should be deployed

3.4 Bootstrapping

After Krake has been installed and runs, the database is still empty. To allow easy insertion of resources during initialisation, a bootstrap script is present, namely: `krake_bootstrap_db`. It is used along with YAML files in which the resources are defined.

Requirements for bootstrapping:

- Krake should be installed;
- the database should be started.

3.4.1 Usage

Workflow

The script is given several files, each with one or several resource definitions. These definitions should follow the structure of the data defined in `krake.data`. See *Structure*.

If the insertion of at least one resource fails, all previous insertions are rolled back. This ensures that the database remains in a clean state in all cases.

The insertion will be rolled back in the following cases:

- the structure of a resource was invalid and its deserialization failed;
- a resource belongs to an API or a kind not supported by the bootstrapping script;

- a resource is already present in the database. This can be overridden using the `--force` flag (see the *force* argument). In this case, a resource already present will be replaced in the database with the currently read definition. In case of rollback, the previous version of the resource will be put back in the database.

Command line

The simplest command is to give one or several files as input, for example:

```
$ krake_bootstrap_db file_1.yaml file_2.yaml
```

Other arguments can be used:

`--db-host` (address): If the database is not present locally, the host or address can be specified explicitly. Default: `localhost`.

`--db-port` (integer): If the database is not present locally, the port can be specified explicitly. Default: `2379`.

`--force`: If set, when the script attempts to insert a resource that is already in the database, the resource will be replaced with its new definition. If not, an error occurs, and a rollback is performed.

The content of the file can alternatively be passed by stdin, using the `-` option:

```
$ cat file_1.yaml | krake_bootstrap_db -
```

This can become very useful when starting the command with a container running Krake. If the file is not present in the container, and you do not want to use a volume, you can still execute the following:

```
$ docker exec -i <krake_container> krake_bootstrap_db - < file_1.yaml
```

3.4.2 Structure

Only resources defined in `krake.data` that are augmented with the `krake.data.persistent` decorator should be inserted with the `krake_bootstrap_db` script.

Each file must have a YAML format, with each resource separated with the `---` separator. The API name, the resource kind and its name must be specified (in the `metadata` for the name).

Thus the minimal resource to add must have the following structure:

```
api: foo
kind: Bar
metadata:
  name: foo_bar
```

This will add a `Bar` object with the name `foo_bar`, with `Bar` defined in the API with name `foo`.

An actual resource would have more values to fill, see the following example with a Krake `Role` and `RoleBinding` definitions:

```
api: core
kind: Role
metadata:
  name: my-role
rules:
- api: 'my-api'
  namespaces:
  - 'my-namespace'
  resources:
  - 'my-resource'
```

(continues on next page)

(continued from previous page)

```
  verbs:
  - list
  - get
---

api: core
kind: RoleBinding
metadata:
  name: my-rolebinding
roles:
- my-role
users:
  - me
```

Danger: The structure of a resource added in the database is checked against the definition of this resource kind. This means that the attributes' name and kind are checked. However, the bootstrapping script does not ensure that the relationships between the resources are valid.

For instance, the `RoleBinding` `my-rolebinding` refers to the `Role` `my-role`. If this role is not in the database, or its name has been misspelled, the bootstrapping script will not detect it, and the database will be inconsistent.

3.4.3 Existing definitions

Some files are already present in the Krake repository with the definitions of different resources.

Authorization

To use the RBAC authorization mode, roles need to be defined, using `Role` objects. They need to be present in the database, and can either be added manually, using the API, or with the bootstrapping:

```
$ krake_bootstrap_db bootstrapping/base_roles.yaml
```

Development and tests

To test the migration, `support/prometheus` or `support/prometheus-mock` script can be used, or simply static metrics. However, in this case, `GlobalMetric` and `GlobalMetricsProvider` objects need to be created. Two bootstrap definition files are present in `support/` for adding Prometheus and static metrics and metrics provider, respectively `prometheus_metrics.yaml` and `static_metrics.yaml`.

They can be easily processed using:

```
$ krake_bootstrap_db support/prometheus_metrics.yaml support/static_metrics.yaml
```

3.5 Security principles

This chapter discusses the different security options supported by Krake, and gives explanation on how to set up Krake securely.

3.5.1 Overview

When sending a request to the API, Krake uses two mechanisms to limit resource access:

- first the *authentication* of the user;
- then, using this information, an *authorization* mechanism describes which resources can be accessed by this user.

Important: There are no user in Krake as actual stored resource. Krake does not manage users, they should be handled by external services (for instance Keystone authentication). Users are identified internally using simple strings. The authentication method ensures that the right string is obtained from a request, and the authorization ensures that the user represented by this string has the right accesses to the resources.

Important: If a Krake component (a controller, or Rok) communicates with the API, the same process is performed. In this case, the user is actually the component itself.

Authentication

To authenticate the user, five different mechanisms can be used: static, Keystone, Keycloak, TLS or anonymous. When a request is received by the API, all the mechanisms enabled in this list will attempt to authenticate the user that sent the request. If the first failed, the second will try, and so on, until all failed, and an HTTP error "Unauthorized (401)" will be sent back to the API. The first that succeeds returns the user that has been authenticated. It is then used during the authorization process.

The order of priority between the authentication mechanisms is as follow (if the mechanisms are enabled):

static: This mechanism should never be used in production. When enabled, this mechanism will authenticate any request as coming from a user with a given username. This username needs to be specified in the API configuration.

Keystone: authenticate incoming requests on the API using an OpenStack Keystone server. A token must first be requested to the Keystone server. This token should then be sent along with any request to the API as the value of the `Authorization` header in the HTTP request. See *Keystone authentication* for more information.

Keycloak: authenticate incoming requests on the API using a Keycloak server. A token must first be requested to the Keycloak server. This token should then be sent along with any request to the API as the value of the `Authorization` header in the HTTP request. See *Keycloak authentication* for more information.

TLS: authenticate incoming requests on the API using the common name attribute of certificates. This name is then used as username. It means TLS needs to be enabled on the API, and thus, on all Krake components. See *Certificate authentication* for more information.

Anonymous: This mechanism should never be used in production. Set using `allow_anonymous` to true in the configuration. If the user has not been authenticated by any previous mechanism, and if anonymous users are allowed, the user will be authenticated as `system:anonymous`.

The authentication mechanisms can be enabled or disabled in the API configuration file, along their specific parameters (see *Authentication and authorization*).

Authorization

The second phase of security is the authorization of an authenticated user. The user is verified against the chosen authorization policy, called authorization mode in the following. If a user has the right to access and perform the

chosen action on the resource currently requested, the request is processed. Otherwise the API returns an HTTP 403 error.

Krake uses three different authorization modes to connect to the API, `always-allow`, `always-deny` and `RBAC`.

`always-allow` all requests are always accepted, for any user;

`always-deny` all requests are always rejected, for any user;

`RBAC` (for Role-Based Access Control) Krake will use roles to decide the resources that a user can access, and the action that this user can perform on these resources.

> **Warning:** The first two modes are only present for testing purposes and should never be used in production. Only `RBAC` should be used in production.

The authorization mode can be chosen in the configuration file (see *Authentication and authorization*).

3.5.2 Keystone authentication

The Keystone authentication uses the OpenStack Keystone service to obtain the identity of a user. The workflow to send a request to the API is as follow if Keystone authentication is enabled:

0. (the user must be registered in Keystone;)
1. the user sends a request to the Keystone server to obtain a token;
2. an HTTP request is sent to the API, with this token used in the header.

Step 1: Kesytone token request

To request a token to the Keystone server, you can use the following example by replacing the values with the corresponding ones for your setup:

```
$ curl -sD - -o /dev/null -H "Content-Type: application/json" \
    http://<keystone_server>/v3/auth/tokens \
    -d '{
      "auth": {
        "identity": {
          "methods": [
            "password"
          ],
          "password": {
            "user": {
              "domain": {
                "name": "<keystone_user_domain_name>"
              },
              "name": "<keystone_username>",
              "password": "<keystone_password>"
            }
          }
        },
        "scope": {
          "project": {
            "domain": {
              "name": "<keystone_project_name>"
            },
```

(continues on next page)

(continued from previous page)

```
            "name": "<keystone_project_domain_name>"
          }
        }
      }
    }'
```

The following example is for the `support/keystone/keystone` script:

```
$ curl -sD - -o /dev/null -H "Content-Type: application/json" \
    http://localhost:5000/v3/auth/tokens \
    -d '{
      "auth": {
        "identity": {
          "methods": [
            "password"
          ],
          "password": {
            "user": {
              "domain": {
                "name": "Default"
              },
              "name": "system:admin",
              "password": "admin"
            }
          }
        },
        "scope": {
          "project": {
            "domain": {
              "name": "Default"
            },
            "name": "system:admin"
          }
        }
      }
    }'
```

You will get an output close to the following, where you can find the expected token:

```
HTTP/1.0 201 CREATED
Date: Tue, 42 Dec 2077 10:02:11 GMT
Server: WSGIServer/1.0 CPython/3.8
Content-Type: application/json
Content-Length: 1234
X-Subject-Token: XXXXXXXXXXXXXXXXXXXXXX   <--- this is the token
Vary: X-Auth-Token
x-openstack-request-id: xxx-xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

From this output, you can obtain your token. A suggestion is to keep it in your shell as environment variable, for instance:

```
$ TOKEN=XXXXXXXXXXXXXXXXXXXXXX
```

Step 2: Keystone token usage

Using your token, you can then communicate with the Krake API:

```
$ curl -k -H "Accept: application/json" -H "Authorization: $TOKEN" \
       <scheme>://<krake_api>/<query>
```

For instance, to get the current authenticated user on Krake installed locally, with TLS support:

```
$ curl --cacert ./tmp/pki/ca.pem -H "Accept: application/json" -H
       "Authorization: $TOKEN" https://localhost:8443/me
```

3.5.3 Keycloak authentication

The Keycloak authentication uses a Keycloak service to obtain the identity of a user. The workflow to send a request to the API is as follow if Keycloak authentication is enabled:

0. (the user must be registered in Keycloak;)
1. the user sends a request to the Keycloak server to obtain a token;
2. an HTTP request is sent to the API, with this token used in the header.

Step 1: Keycloak token request

Query

To request a token to the server, multiple ways are supported by Keycloak. If the server has been set up for direct access grants, you can use the following example by replacing the values with the corresponding ones for your setup:

```
$ curl -s http://localhost:9080/auth/realms/krake/protocol/openid-connect/token \
    -d 'grant_type=password' \
    -d 'username=<username>' \
    -d 'password=<password>' \
    -d 'client_id=<client_name>' \
    -d 'client_secret=<client_secret>'
```

For the `support/keycloak` script, you can use the following command to get a token:

```
$ support/keycloak token
```

Internally, something similar to the following is used, with all values set by the script:

```
$ curl -s http://localhost:9080/auth/realms/krake/protocol/openid-connect/token \
    -d 'grant_type=password' \
    -d 'username=krake' \
    -d 'password=krake' \
    -d 'client_id=krake_client' \
    -d 'client_secret=AVeryCoolAndSecureSecret'
```

Response

Using the cURL queries, you will get a JSON with the following structure:

```
{
    "access_token":"XXXXXXXXXXXXXXXX",
    "expires_in":60,
```

(continues on next page)

(continued from previous page)

```
    "refresh_expires_in":1800,
    "refresh_token":"<refresh_token>",
    "token_type":"bearer",
    "not-before-policy":0,
    "session_state":"9c22a6df-0997-4d3d-a540-239f85346008",
    "scope":"profile email"
}
```

From this output, you can obtain your token from the `access_token` field. A suggestion is to keep it in your shell as environment variable, for instance:

```
$ TOKEN=XXXXXXXXXXXXXXXXXXXXX
```

With the `support/keycloak` direct command, you get the token directly, thus you could simply use:

```
$ TOKEN=$(support/keycloak token)
```

Step 2: Keycloak token usage

Using your token, you can then communicate with the Krake API:

```
$ curl -k -H "Accept: application/json" -H "Authorization: $TOKEN" \
        <scheme>://<krake_api>/<query>
```

For instance, to get the current authenticated user on Krake installed locally, with TLS support:

```
$ curl --cacert ./tmp/pki/ca.pem -H "Accept: application/json" -H
        "Authorization: $TOKEN" https://localhost:8443/me
```

3.5.4 Certificate authentication

With the TLS support enabled on the API configuration, the requests to the API can only be performed using HTTPS. This allows Krake to obtain information about the sender through the certificates. Especially, Krake can use the common name to identify the user that sent the request.

This authentication mechanism should always be used in a production environment. It also allows the authentication of the Krake components. The scheduler, the garbage collector or any other controller should have a certificate with a specific common name. This name can then be used along with the `RBAC` mode and a specific `RoleBinding` to allow the controller to access the resources it needs.

Important: With TLS support, all Krake components will use certificates with their corresponding key. All components (API, controllers and rok) must use the same CA, and the certificates they use for communication must also be signed using this CA.

Note: If an external endpoint is specified in the Kubernetes controller configuration for the *complete* hook, then this host must also be specified in the certificate of the API.

3.5.5 RBAC Authorization

The Role-Based Access Control (or RBAC) is a model of resource access. Each user is given one or several roles, and each role has access to one or several resources, and/or actions.

When RBAC is enabled, roles need to be defined and bound to users using respectively `Role` and `RoleBinding` core API objects. They have their own endpoints for creation, update, deletion... (`/core/roles` and `/core/rolebindings` respectively).

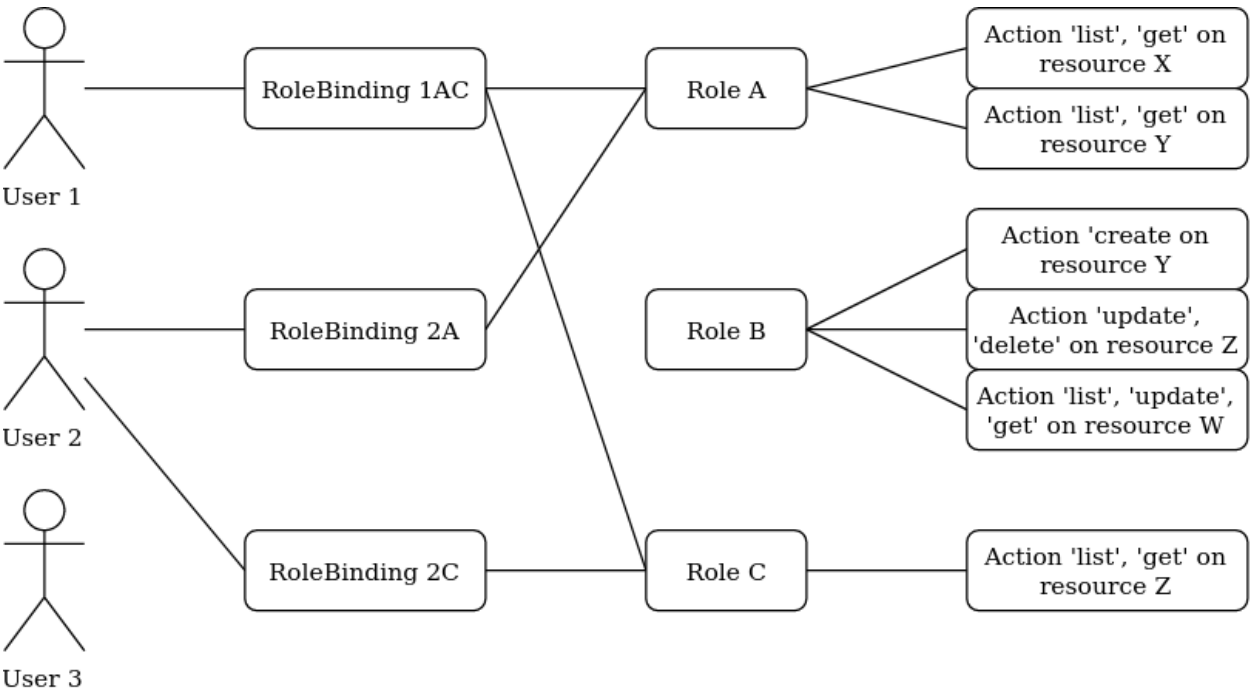
Role bindings

The `RoleBinding` objects defines a connection between one or several roles to one or several users.

Roles

A `Role` defines different rules: each rule describes which resource can be accessed by a user with this role, and which action can be performed. The `Role` can then be applied to several users, which is the purpose of `RoleBinding` objects.

Example

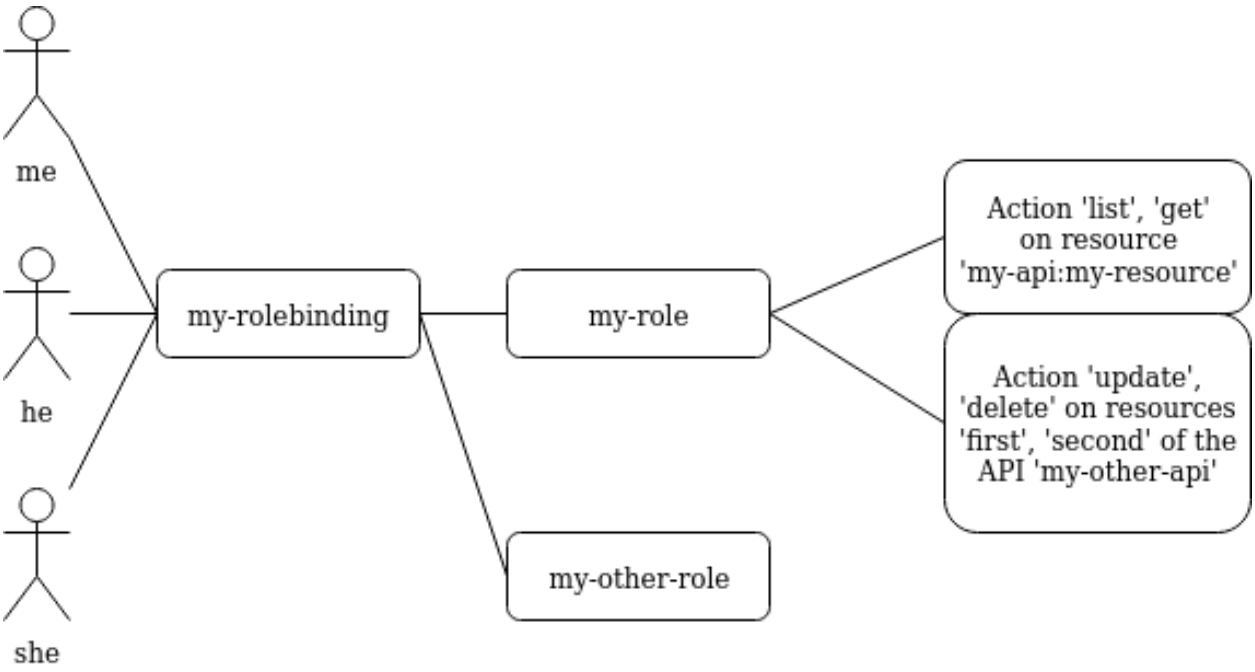


In the previous example, the user 1 and 2 have both been given the roles A and C. It means they can both get and list the resources X, Y and Z.

Let's now say we want to have the following minimal example:

```
api: core
kind: Role
metadata:
  name: my-role
```

(continues on next page)



(continued from previous page)

```
rules:
- api: 'my-api'
  resources:
  - 'my-resource'
  namespaces:
  - 'my-namespace'
  verbs:
  - list
  - get
- api: 'my-other-api'
  resources:
  - 'first'
  - 'second'
  namespaces:
  - 'my-namespace'
  verbs:
  - update
  - delete
```

In the above example definition of a `Role`, a user with this role can:

- list or read the `my-resource` resources defined in the `my-api` API, that belong to the `my-namespace` namespace (first rule);
- update or delete the `first` and `second` resources defined in the `my-other-API` API, also in the `my-namespace` namespace (second rule).

```
api: core
kind: RoleBinding
metadata:
  name: my-rolebinding
roles:
- my-role
```

(continues on next page)

(continued from previous page)

```
- my-other-role
users:
- me
- he
- she
```

In the above example, the `RoleBinding` object binds the `my-role` and the `my-other-role` to the users `me`, `he` and `she`.

3.5.6 Security Guidelines

Warning: DISCLAIMER: The steps described in this chapter do not ensure a fully secure Krake infrastructure. They are the minimal security steps that are recommended. An actual fully secure setup need general security measures on all its components and on the setup itself, not only for the Krake infrastructure

This section is a guide that describes all the steps to create a minimal secure Krake infrastructure.

What you need:

- Krake installed;
- the Krake repository (optional);
- a certificate authority (CA) and at least five certificates and their respective keys (signed with this CA). To follow this guide easily, the common names of the certificates shall be:
  - `system:gc`
  - `system:scheduler`
  - `system:kubernetes`
  - `system:magnum`
  - `system:infrastructure`
  - `system:admin`
  - `system:complete-signing`
  - `system:shutdown-signing`
  - an additional certificate is necessary for the API.

  These names are the ones present in the bootstrapping file called `base_roles.yaml`. They can naturally be modified to follow your needs.

The `support/pki` script can also generate them for testing purpose, example:

```
$ support/pki system:admin
```

The certificate with `system:complete-signing` will be used for signing new certificates, thus would need to be set for signing purposes:

```
$ support/pki system:complete-signing --intermediate-ca
```

The certificate with `system:shutdown-signing` will be used for signing new certificates, thus would need to be set for signing purposes:

```
$ support/pki system:shutdown-signing --intermediate-ca
```

If Krake is not deployed locally, you also need to set its external endpoint as TLS subject alternative names, for instance:

```
$ support/pki system:api-server --host 1.2.3.4 --host example.com
```

Configuration of the API

The first step is to configure the API to use the right authentication and authorization modes.

Configuration of the authentication:

First, disable the static and anonymous authentications in the API configuration:

```
authentication:
  allow_anonymous: false

  #...
  static:
    enabled: false
    name: system:admin
```

Then, enable the TLS support on the API:

```
tls:
  enabled: true
  cert: <path_to_your_certificate>
  key: <path_to_your_key>

  client_ca: <path_to_your_client_ca>
```

If you want to use Keystone or Keycloak authentication additionally, you should set the configuration as well:

```
authentication:
  # ...

  strategy:
    # Keystone authentication
    keystone:
      enabled: true
      endpoint: <your_keystone_endpoint>

    # Keycloak authentication
    keycloak:
      enabled: true
      endpoint: <your_keystone_endpoint>
      realm: <your_keycloak_realm>
```

Krake contains an example Keystone server under `support/keystone/Dockerfile`. This is a docker file, which creates an image with a secure Keystone instance, that can be accessed over HTTPS.

Configuration of the authorization:

To set the `RBAC` authorization mode, change the following line in the API configuration:

```
authorization: RBAC
```

Configuration of the Controllers

You need to enable the TLS support on all controllers:

```
tls:
  enabled: true
  client_ca: <path_to_your_client_ca>
  client_cert: <path_to_your_client_cert>
  client_key: <path_to_your_client_key>
```

The API endpoint must be modified to use HTTPS:

```
api_endpoint: https://<endpoint>
```

This certificate must indicate a common name only used by the current controller. Let's refer to it as `system:<controller>` as an example. Using TLS authentication, `system:<controller>` will be the username of the Controller every time this controller will connect to the API, see *Authentication*

When using bootstrapping, the "username" of the controllers must be adapted to correspond to the ones in the `RoleBinding` objects added in the database. See *Database bootstrapping*.

If the bootstrapping file present in the repository is used (`base_roles.yaml`), the common names of the controller certificates must be:

- `system:gc` for the garbage collector;
- `system:scheduler` for the scheduler;
- `system:kubernetes` for the Kubernetes controller;
- `system:complete-signing` for the signing certificate of the "complete" hook, see *Complete*.
- `system:shutdown-signing` for the signing certificate of the "shutdown" hook, see *Shutdown*.
- `system:magnum` for the Magnum controller.

Configuration of rok

```
api_url: https://<endpoint>   # must use HTTPS
user: <rok_user>

tls:
  enabled: true
  client_ca: <path_to_your_client_ca>
  client_cert: <path_to_your_client_cert>
  client_key: <path_to_your_client_key>
```

The common name used by the certificate must match the one from `<rok_user>`. This name will be used as username.

If the bootstrapping file present in the repository is used (`base_roles.yaml`), the certificate used by the administrator must have `system:admin` as common name, and `<rok_user>` must then match it.

Database bootstrapping

For the RBAC authorization mode to work, `Role` and `RoleBinding` objects need to be put in the database.

They can be either added manually using the command line, or more simply added by using bootstrapping (see *Bootstrapping*). The roles for the Krake components and the administrator are defined already in `bootstrapping/base_roles.yaml`. Thus they can all be added with:

```
$ krake_bootstrap_db bootstrapping/base_roles.yaml
```

When using the `base_roles.yaml`, the usernames in the `RoleBinding` for the controllers must match the ones used by the certificates.

For instance for the garbage collector, if the `RoleBinding` is defined like this:

```
api: core
kind: RoleBinding
metadata:
  name: rolebinding:system:gc
roles:
- role:system:gc
users:
- system:gc
```

it means that the certificate common name for the garbage collector must be `system:gc`. It is probably easier to adjust the `base_roles.yaml` to match your needs.

Additional roles and role bindings can also be added to the database using the same bootstrapping method, by modifying the `base_roles.yaml`, or by writing another file and bootstrapping it into the database.

Administrator

The role `role:system` added in the `base_roles.yaml` corresponds to an administrator role, and the role binding `rolebinding:system` allows a user called `system:admin` full access to all Krake resources from all APIs. These two can naturally be modified if the administrator should have another name.

Important: Note that if no administrator user is created, `Role` and `RoleBinding` objects cannot be created through the API, but must be added to the database directly.

3.5.7 CORS

The Cross-origin resource sharing (CORS) mechanism was enabled on Krake but the fields are set to be quite non-restrictive. By default, the `Access-Control-Allow-Origin` is set to `*`. With this setup, sending request through a browser could be dangerous. A user could first connect to a valid website with some allowed authentication token and send requests to Krake. Then the user goes on a malicious website, which may be able to reuse the token, as the default value accepts any origin, so any URL. To prevent this situation, the value for the `Access-Control-Allow-Origin` field can be set for the Krake API, see the *Authentication and authorization* part of the configuration.

CHAPTER 4

Developer Documentation

This section of the documentation is dedicated to all contributors of the project. It describes the overall system architecture, explains the core concepts of the system and development principles that should be followed when contributing.

Furthermore, the layout of the repository is explained and a complete Python API reference of all modules is provided.

4.1 Architecture

This chapters gives a high-level overview about the software architecture of Krake. The following figure gives an overview about the components of Krake. The components are described in more detail in the following sections.

4.1.1 API

The API is the central component of Krake. It holds the global state of the system. Krake uses an abstraction for real world objects – e.g. Kubernetes clusters or clouds (e.g. OpenStack) – managed or used by it. The objects are represented as RESTful HTTP resources called *API resources*. These resources are stored in an associated etcd database. Each resource is a nested JSON object following some conventions that can be found in section *API Conventions*.

The API is *declarative*: instead of sending commands one-by-one to the infrastructure, the user just defines a desired end state, telling the infrastructure exactly how it should look like. Then, the *Control Plane* works in sync with the infrastructure to find the best way to get there. This means that the actual *control flow* is not exposed to the user.

4.1.2 Control Plane

The control plane is responsible for bringing the declarative API to life: it synchronizes the declared desired state of a API resource with the managed real world object (see also *Control Plane* on the concepts chapter).

The control plane consists of a set of *controllers*. Normally, one controller is responsible for one kind of resource, e.g. the Kubernetes Application controller manages Kubernetes Application resources. Only API resources with a changing state are managed by a controller.

System-level tasks are also handled by controllers:

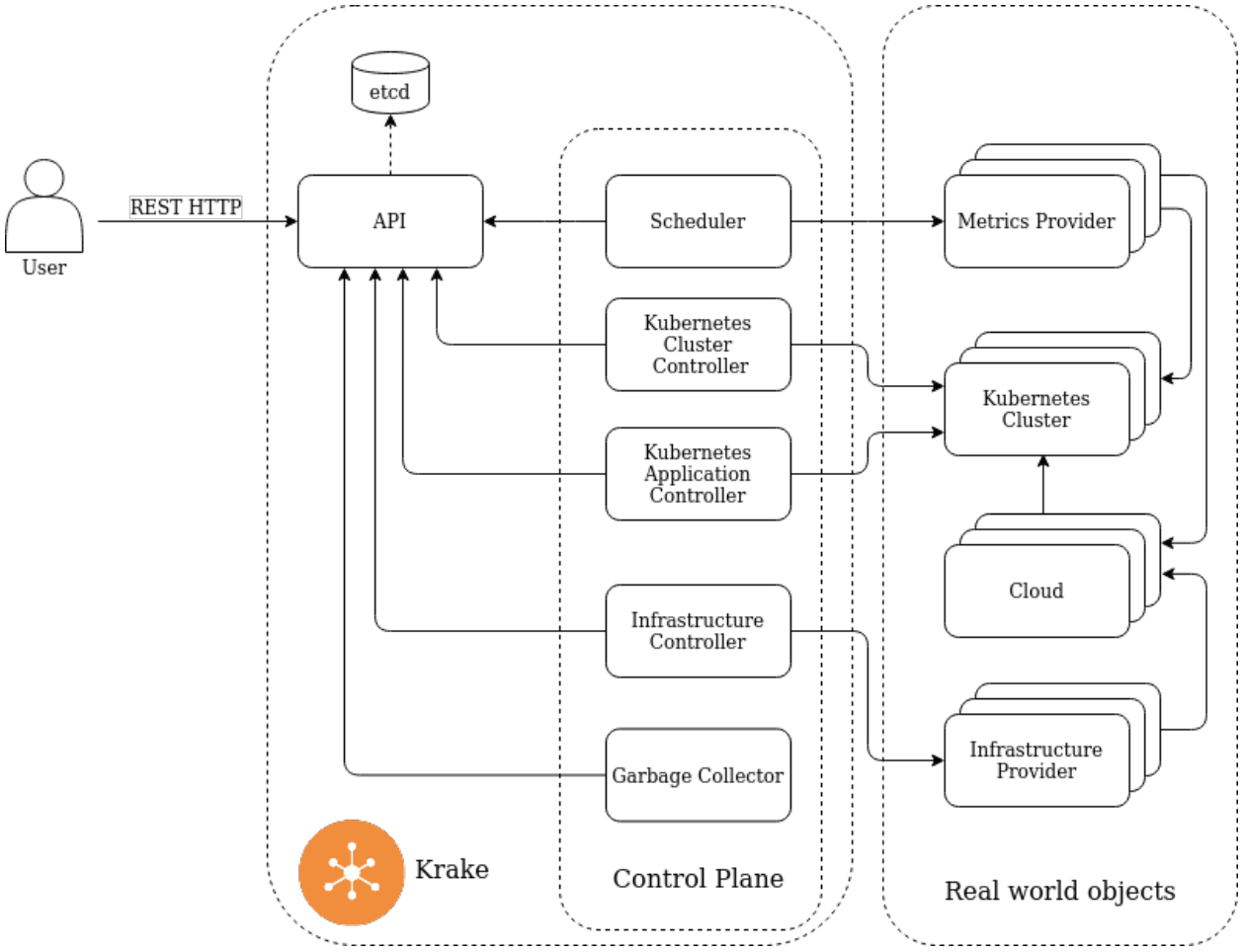


Fig. 1: Krake components

Garbage Collector Resources can depend on other resources. If an API resource is deleted, dependents of the resource are also deleted automatically. This is called *cascading deletion*. The garbage collector is the controller responsible for executing the cascading deletion (see *Garbage Collection* for more details).

Infrastructure Controller The infrastructure controller performs life-cycle management of the real-world Kubernetes clusters. (see *Infrastructure Controller* for more details).

Scheduler The scheduler is a very important controller responsible for binding applications – high-level API resources for executing workloads – to a platform managed by Krake.

Tip: For example, the scheduler binds Kubernetes applications to Kubernetes clusters or selects clouds (e.g. OpenStack) for creating new Kubernetes clusters.

The scheduler makes its decision based on a set of metrics provided by external metrics providers (see *Scheduling* for more details) as well as the availability of the clusters. The decisions are periodically reevaluated, which could potentially lead to *migration* of applications.

Control Loop

The following figure describes the basic control loop that is executed by any controller.

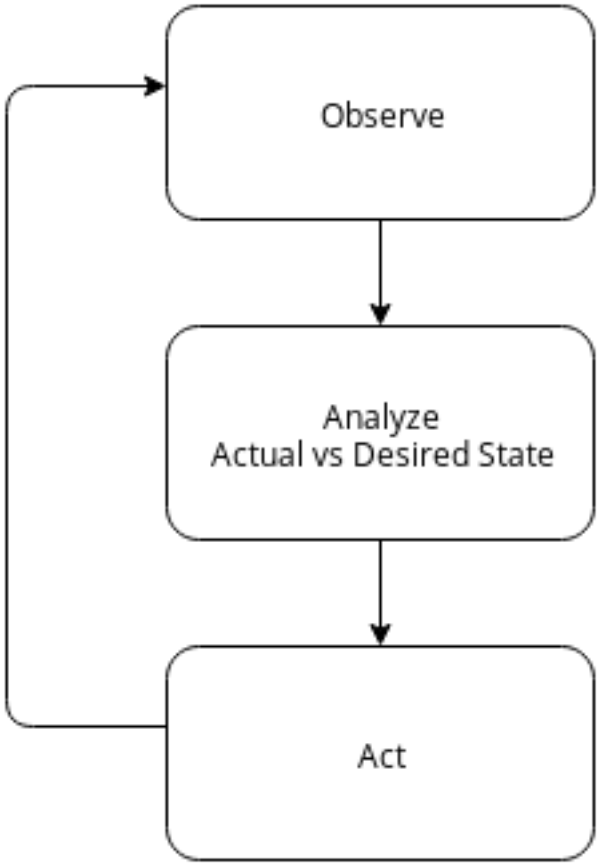


Fig. 2: Operator Pattern

The observation of an API resource is done via *watching*, a long running HTTP request to the API notifying about changes of resources:

```
$ curl http://localhost:8080/kubernetes/namespaces/testing/applications?watch
{"type": "UPDATED", "object": {"metadata": ..., "spec": ..., "status": ...}}
{"type": "DELETED", "object": {"metadata": ..., "spec": ..., "status": ...}}
{"type": "UPDATED", "object": {"metadata": ..., "spec": ..., "status": ...}}
...
```

Controller will read this feed, evaluate differences between the desired state and the state of the managed real world object, act accordingly to this difference and update the status of the resource.

4.2 Concepts

Krake is heavily inspired by the concepts of Kubernetes. If you are familiar with the internal mechanisms of Kubernetes you should find many similarities within Krake.

4.2.1 Overview

The central service of Krake is a RESTful HTTP API. The API is structured in groups of APIs covering different technologies, e.g. `core` for the core functionalities or `kubernetes` for Kubernetes-specific features. Each API comprises multiple kinds of resources, e.g. the `kubernetes` API contains *Application* or *Cluster* resources. The resources are used to describe the *desired state*. The user can update the desired state by updating the resource via simple PUT requests to the API.

In concept, every resource is handled by a *Controller*. The responsibility of a controller is to bring the described *desired state* of a resource in sync with the real world state. Some of the resources act as mere data bags, e.g. *kubernetes/Cluster* resources simply describe how to connect to an existing Kubernetes cluster. These resources do not have a corresponding controller because no logic is needed for syncing desired and real world state.

4.2.2 API Conventions

Krake uses abstractions for real world resources managed by Krake, e.g. Kubernetes clusters spawned on top of an OpenStack deployment. These abstractions are represented as API resources encoded as nested JSON objects.

Metadata

Every resource *MUST* have the following metadata in a nested field called `metadata` with the following structure:

namespace A namespace is used to isolate access resources. Normally, a user does only get access to a specific namespace. See *Authentication and Authorization* for more details. This field is immutable which means a resource cannot be migrated to another namespace.

name A string uniquely identifying a resource in its namespace. This name is used in URLs when operating on an individual resource. This field is immutable.

uid A unique string in time and space used to distinguish between objects of the same name that have been deleted and recreated. This field is immutable.

finalizers A list of strings that can be added by controllers to block the deletion of the resource in order to do some clean up work (finalizing). A resource *MUST* not be deleted if there is at least one finalizer.

Controllers *SHOULD* process only finalizers that were added by them and that are at the tail of the list. This ensures a strict finalizing order.

created The timestamp when the resource was created. This field is immutable.

modified The timestamp when the `spec` or `metadata` field of the resource was changed.

deleted The timestamp when the resource was deleted. If this field is set, the resource is in the *in deletion* state. This transition is irreversible. In this state, no changes to the resource are allowed except removing items from `finalizers` and updating the `status`. If `finalizers` is empty and the resource is *in deletion* it will be removed from the database. See *Garbage Collection* for more details.

Spec and Status

By convention, the Krake API distinguishes between *desired state* of a resource – a nested field called `spec` – and its *real world state* – a nested field called `status`.

Every resource representing a real world object managed by Krake *SHOULD* have a field called `spec`. If the state of the represented object cannot change, the resource *MAY* have a `spec` field only which *MAY* be renamed to a more appropriate name.

etcd

Internally, the Krake API uses etcd – a distributed and reliable key-value store – as persistence layer. But this is considered an implementation detail and no etcd-specific mechanisms are exposed via the REST API. This means that the underlying database could be potentially replaced in the future if the requirements of the project change. The "killer" feature of etcd is the watching of keys and prefixes for changes.

Note: The distributed nature of etcd and its built-in support for observing changes for specific keys were the main motivation why Krake switched from a SQL-based persistence layer to etcd.

4.2.3 Control Plane

The API does not implement control logic. The task of reconciling between *desired state* and *real world state* is done by so-called controllers. Controllers are independent services watching API resources and reacting on changes. The set of all controllers forms the *Control Plane* of Krake.

Controllers communicate with the API server: the desired state is fetched from the API and status updates are pushed to the API. In theory, controllers can be programmed with any technology (programming language) capable of communicating with a REST HTTP interface.

Note: The first system architecture of Krake was event-based using message queuing (RabbitMQ). The main issue with event-driven systems is that the they get out-of-sync if a message gets lost. Hence, a lot of effort is involved to make sure that no message loss occurs.

On the other hand, level-based logic operates given a desired state and the current observed state. The functionality is resilient against loss of intermediate state updates. Hence, a component can recover easily from crashes and outages, which makes the overall system more robust. This was the motivation for moving from an event-based system with message queuing to a level-based system with reconciliation.

4.2.4 Authentication and Authorization

Access to the API is provided through a two-phased process.

Authentication Each request to the Krake API is authenticated. Authentication verifies the identity of the user. There are multiple authentication providers and the API can be extended by further authentication mechanisms. If no identity is provided, the request is considered to be *anonymous*. For internal communication between controllers and API, TLS certificates *SHOULD* be used.

Authorization After the identity of a user is verified, it needs to be decided if the user has permission to access a resource.

Krake implements a simple but powerful role-based access control (RBAC) model. The `core` API provides `Role` resources describing access to specific operations on specific resources potentially in specific namespaces. A user is assigned to a role by another `core` resource called `RoleBinding`.

Roles in Krake are **permissive** only. There is no way to deny access to a resource through a role. At least one role a user is bound to needs to allow access to the requested resource and operation. Otherwise access is denied.

4.3 Directories

The Krake repository contains several main directories, which will be described here.

ansible Contains all the configuration and the playbooks to install Krake using Ansible.

docker Contains all file to start Krake in a Docker infrastructure.

docs Contains the source files for Krake documentation.

infra Contains scripts to deploy Krake. Not up-to-date.

krake Contains the source code and the unit tests for the Krake application. More details are given in the Krake Reference.

rok Contains the source code and the unit tests for the Rok command. More details are given in the Rok Reference.

support Contains the utility scripts to create a simple local test environment.

4.4 Design Principles

This section contains a number of principles that should be followed when extending Krake. The principles are very similar to the Kubernetes design principles.

4.4.1 API

See also *API Conventions*.

- All APIs should be declarative.
- API resources should be *complementary* and *composable*, not opaque wrappers.

> **Note:** For example, a Kubernetes cluster could be created on top of a managed OpenStack project.

- The control plane should be *transparent* – there are no hidden internal APIs.
- Resource status must be *completely reconstructable by observation*. Any history kept (caching) must be just an optimization and not required for correct operation.

4.4.2 Control Logic

- Functionality must be level-based, meaning the system must operate correctly given the desired state and the current/observed state, regardless of how many intermediate state updates may have been missed. Event/Edge-triggered behavior must be just an optimization.

Note: There should be a CAP-like theorem regarding the trade-offs between driving control loops via polling or events about simultaneously achieving *high performance*, *reliability*, and *simplicity* – pick any two.

- Assume an open world: continually verify assumptions and gracefully adapt to external events and/or actors.

Tip: For example, Krake allows users to kill Kubernetes resources under control of a Kubernetes application controller; the controller just replaces the killed resource.

- Do not assume any state transition or state that cannot be determined by observation.
- Do not assume a component's decisions will not be overridden or rejected, nor for the component to always understand why.

Tip: For example, etcd may reject writes. The scheduler may not be able to schedule applications. A Kubernetes cluster may reject requests.

- Retry, but back off and/or make alternative decisions.
- Components should be *self-healing*.

Tip: For example, if some state must be kept, e.g. cached, the content needs to be periodically refreshed, so that if an item does get incorrectly stored or a deletion event is missed, the kept state will be soon synchronized, ideally on timescales that are shorter than what will attract attention from humans.

- Component behavior should *degrade gracefully*. Actions should be prioritized such that the most important activities can continue to function even when overloaded and/or in states of partial failure.

4.4.3 Architecture

- Only the API server communicate with etcd, and no other components, e.g. scheduler, garbage collector, etc.
- Components should continue to do what they were last told in the absence of new instructions, e.g. due to network partition or component outage.
- All components should keep all relevant state in memory all the time. The API server writes through to etcd, other components write through to the API server, and they watch for updates made by other clients.
- Watch is preferred over polling.

4.4.4 Extensibility

- All components should be replaceable. This means there is no strong coupling between components.

Tip: For example, the different scheduler should be usable without any changes in another component.

- Krake is extended with new technologies/platforms by adding new APIs.

4.4.5 Availability

Note: High-availability (HA) is about removing **single point of failure** (SPOF).

- HA is achieved by service replication.

Todo: It needs to be decided on which level replication is introduced.

Coarse grained Replicate "Krake master" with all included components, e.g. API server, controllers etc.

Fine grained Replicate single components. If a component is stateful – relevant state should be kept in memory as stated in section *Architecture* – the components should follow an active-passive principle where only one replica of a component is active at the same time. A etcd lease may be a good option for this but only the API should have direct access to etcd. A solution for this would be to introduce special API endpoints for electing a leader across multiple replicas.

4.4.6 Development

- Self-hosting of all components is the goal.
- Use standard tooling and de facto standards of the Python ecosystem.
- Keep dependencies as small as possible, but do not reinvent the wheel.

4.5 Scheduling

This part of the documentation presents the Krake scheduling component and how the Krake resources are handled by the scheduling algorithm.

The Krake scheduler is a standalone controller that processes the following Krake resources:

- Application
- Cluster
- Magnum cluster (Warning: Due to stability and development issues on the side of Magnum, this feature isn't actively developed anymore.)

The scheduler algorithm selects the "best" backend for each resource based on metrics of the backends and the resource's constraints and specifications. The following sections describe the application and Magnum cluster handlers of the Krake scheduler.

4.5.1 Application handler

The application handler is responsible for scheduling and rescheduling (automatic migration) of the applications with non-deleted and non-failed states. Applications with deleted or failed state are omitted from the scheduling. Currently, the application handler considers every Kubernetes cluster.

Scheduling of Applications

- At first, the scheduler checks, if the clusters that will be considered and filtered to host the application, are even ONLINE. If a cluster isn't reachable, it is not considered in the scheduling process.
- The application handler evaluates if all constraints of an application match the available Kubernetes cluster resources. The application constraints define restrictions for the scheduling algorithm. Currently, the custom resources constraint, the cluster label constraint and the metric constraint are supported, see *Constraints*. This is a first **filtering** step.
- Selected Kubernetes clusters could contain metrics definition. If the cluster contains metrics definition, the application handler fetches metric values from the corresponding metrics providers which should be defined in the metric resource specification, see *Metrics and Metrics Providers*.
- Then, the score for each Kubernetes cluster resource is computed. The cluster score is represented by a decimal number and is computed from Kubernetes cluster stickiness and metric values and weights. More information on stickiness in the *Stickiness* section. The Kubernetes clusters with defined metrics are preferred. It means clusters which are not linked to any metric have a lower priority than the ones with linked metrics:
  - If there are no cluster with metrics: the score is only computed using the stickiness;
  - If there are clusters with metrics: the clusters without metrics are filtered out and the scores of the ones with metrics are computed and compared.

  This step is a **ranking** step.
  - The score formula for a cluster without metrics is defined as follows:

$$score = sticky_{value} \cdot sticky_{weight}$$

  - The score formula for a cluster with *n* metrics is defined as follows:

$$score = \frac{(sticky_{value} \cdot sticky_{weight}) + \sum_{i=1}^{n} metric_{value_i} \cdot metric_{weight_i}}{sticky_{weight} + \sum_{i=1}^{n} metric_{weight_i}}$$

- The application is scheduled to the cluster with the highest score. If several clusters have the same score, one of them is chosen randomly.

Rescheduling (automatic migration) of Applications

- Applications that were already scheduled are put in the scheduler controller queue again, to be rescheduled later on. Applications will go through the scheduling process again after a certain interval, which is defined globally in the scheduler configuration file, see *Configuration* (defaults to *60s*). This parameter is called *reschedule_after*. It allows an application to be rescheduled to a more suitable cluster if a better one is found.

Stickiness

Stickiness is an extra metric for clusters to make the application "stick" to it by increasing its score. Stickiness extra metric is defined by its value and configurable weight. It represents the cost of migration of an Application, as it is added to the score of the cluster on which the Application is currently running.

If the value is high, no migration will be performed, as the updated score of the current cluster of the Application will be too high compared to the score of the other clusters. If this value is too low, or if this mechanism was not present, any application could be migrated from just the slightest change in the clusters score, which could be induced by small changes in the metrics value. Thus the stickiness acts as a threshold: the changes in the metrics values has to be higher than this value to trigger a rescheduling.

The stickiness weight is defined globally in the scheduler configuration file, see *Configuration* (defaults to *0.1*). If the application is already scheduled to the passed cluster, a cluster stickiness is *1.0* multiplied by the weight, otherwise *0*.

Application Handler's workflow:

The following figure gives an overview about the application handler of Krake scheduler.

Special note on updates:

kube_controller_triggered:

This timestamp is used as a flag to trigger the Kubernetes Controller reconciliation. Together with `modified`, it's allowing correct synchronization between Scheduler and Controller.

It is updated when the chosen `Cluster` has changed, or once after the update of an Application triggered its rescheduling, even if this did not change the scheduled cluster. The second case mostly occurs when a user updates it through the API.

This timestamp is used to force an `Application` that has been updated by a user to be rescheduled before the changes are applied by the Kubernetes Controller. Without this mechanism, the `Application` may be updated, but rescheduled somewhere else afterwards.

The actual workflow is the same as the one explained in the schema above. However, there is an additional interaction with the Kubernetes Controller:

- The user updates the `Application` `my-app` on the API:

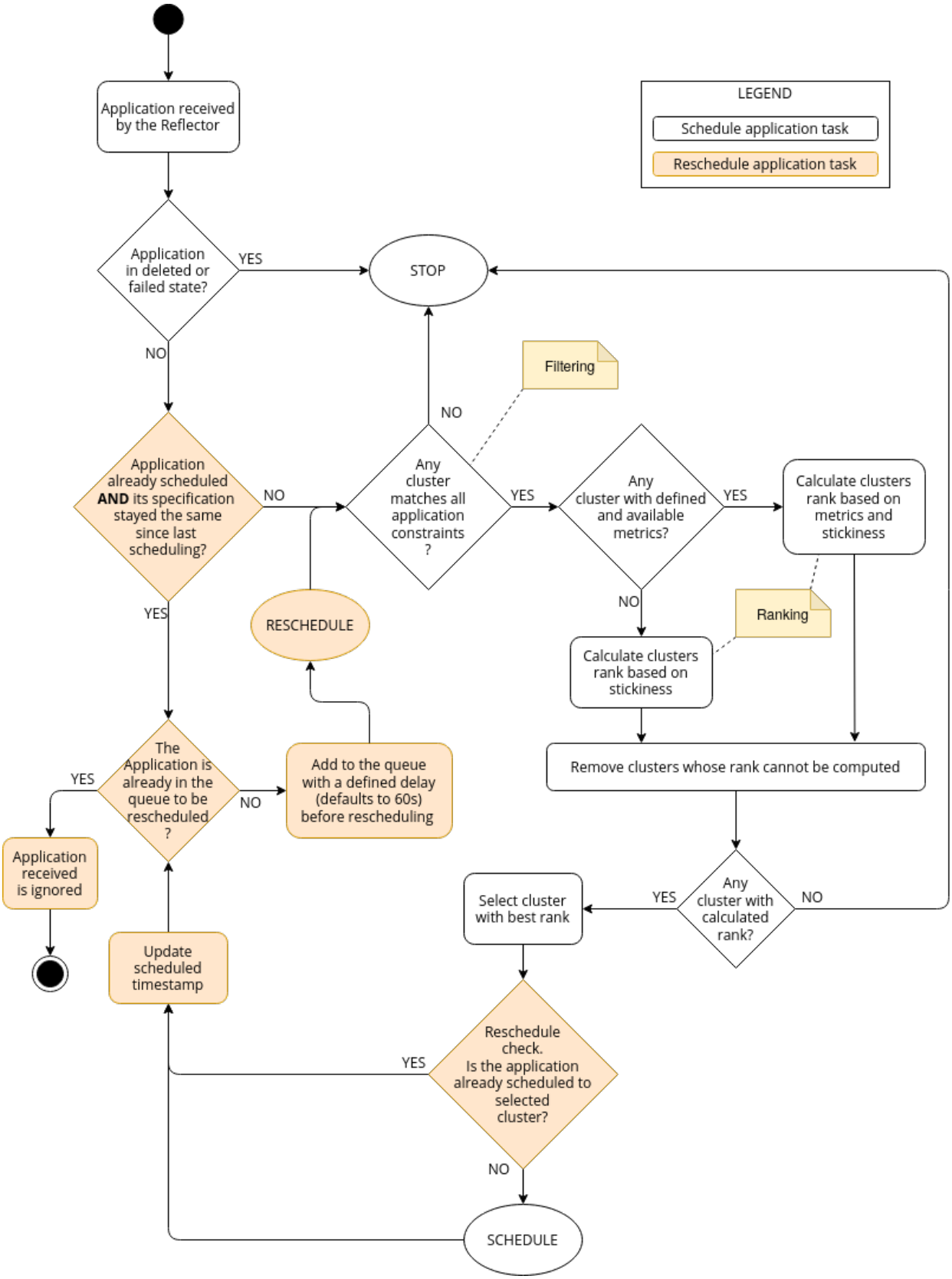
  `my-app`'s `modified` timestamp is **higher** than the `kube_controller_triggered` timestamp;

- The Kubernetes Controller rejects the update on `my-app` in this case;
- The Scheduler accepts the update on `my-app` and chooses a cluster for the updated `my-app`;
- as the cluster changed, the `kube_controller_triggered` timestamp is updated;

  `my-app`'s `modified` timestamp is **lower** than the `kube_controller_triggered` timestamp;

- the updated `my-app` is rejected by the Scheduler because of this comparison;
- the updated `my-app` is accepted by the Kubernetes Controller;
- the actual updates of the `Application` are performed by the Kubernetes Controller if needed.

When the Application is rescheduled, if the selected cluster did not change, then the `kube_controller_triggered` timestamp is updated only if the rescheduling was triggered by an update of the Application. If the Application is rescheduled on the same cluster automatically, then the timestamp is not


LEGEND
Schedule application task
Reschedule application task
Application received by the Reflector
Application in deleted or failed state?
YES
STOP
NO
Application already scheduled AND its specification stayed the same since last scheduling?
NO
Any cluster matches all application constraints ?
NO
Filtering
YES
Any cluster with defined and available metrics?
YES
Calculate clusters rank based on metrics and stickiness
NO
Ranking
Calculate clusters rank based on stickiness
YES
RESCHEDULE
Remove clusters whose rank cannot be computed
The Application is already in the queue to be rescheduled ?
YES
NO
Add to the queue with a defined delay (defaults to 60s) before rescheduling
Application received is ignored
Any cluster with calculated rank?
YES
NO
Select cluster with best rank
Update scheduled timestamp
Reschedule check. Is the application already scheduled to selected cluster?
YES
NO
SCHEDULE

updated. This prevents an update of each Application on each automatic rescheduling, which would need to be handled by the Kubernetes controller.

To sum up, the `kube_controller_triggered` timestamp represent the last time this version of the Application was scheduled by the Scheduler.

scheduled:

The `scheduled` timestamp expresses the last time the scheduling decision changed for the current resource. This timestamp does not correspond to the time where the Application was deployed on the new cluster, just the time where the scheduler updated, on the Application, the reference to the cluster where it should be deployed. It is actually updated during a call from the scheduler to the API to change the binding of the Application.

This timestamp is however not updated if an update of its Application did not lead to a rescheduling, just a re-deployment.

4.5.2 Cluster handler

The Cluster handler is responsible for scheduling Kubernetes Clusters to the best cloud backend (currently Krake supports OpenStack as a cloud backend). The Cluster handler should process only Clusters that are not bound to any Cloud, have non-deleted/non-failed state and also the Clusters should not contain the kubeconfig file in their *spec*. If the Cluster contains the kubeconfig file in its *spec* it is considered as an existing cluster which was registered or created by Krake and therefore should be ignored by the Cluster handler.

Scheduling of Clusters

- The Cluster handler evaluates if all the constraints of a Cluster match the available cloud resources. The Cluster constraints define restrictions for the scheduling algorithm. Currently, Cloud label and metrics constraints are supported, see *Constraints*.
- If the selected Cloud resources contain metric definitions, the Cluster handler fetches metric values from the corresponding metrics providers which should be defined in the metric resource specifications, see *Metrics and Metrics Providers*.
- Then, the score for each Cloud resource is computed. The Cloud score is represented by a decimal number and is computed from metric values and weights. If a given Cloud does not contain any metric definition, its score is set to *0*. Therefore, the Clouds with defined metrics are preferred:
  - If there are no Clouds with metrics: the score is 0 for all Clouds.
  - If there are Clouds with metrics: the Clouds without metrics are filtered out and the scores of the ones with metrics are computed and compared.

  This step is a **ranking** step.
  - The score formula for a Cloud without metrics is defined as follows:

$$score = 0$$

  - The score formula for a Cloud with *n* metrics is defined as follows:

$$score = \frac{\sum_{i=1}^{n} metric_{value_i} \cdot metric_{weight_i}}{\sum_{i=1}^{n} metric_{weight_i}}$$

- The Cluster is scheduled to the Cloud with the highest score. If several Clouds have the same score, one of them is chosen randomly.

The following figure gives an overview about the Cluster handler of the Krake scheduler.

4.5.3 Magnum cluster handler

Warning: Due to stability and development issues on the side of Magnum, this feature isn't actively developed anymore.

The Magnum cluster handler is responsible for scheduling Magnum clusters to the best OpenStack project. The Magnum cluster handler should process only Magnum clusters that are not bound to any OpenStack project and have non-deleted state. Currently, the Magnum cluster handler considers every OpenStack project.

Scheduling of Magnum clusters

- The Magnum cluster handler evaluates if all the constraints of a Magnum cluster match the available OpenStack project resources. The Magnum cluster constraints define restrictions for the scheduling algorithm. Currently, only the OpenStack project label constraints are supported, see *Constraints*. This is a first **filtering** step.
- Selected OpenStack project resources could contain metric definitions. If the OpenStack project contains metrics definition, the Magnum cluster handler fetches metric values from the corresponding metrics providers which should be defined in the metric resource specifications, see *Metrics and Metrics Providers*.
- Then, the score for each OpenStack project resource is computed. The OpenStack project score is represented by a decimal number and is computed from metric values and weights. If a given OpenStack project does not contain metric definition, its score is set to *0*. Therefore, the OpenStack projects with defined metrics are preferred:
  - If there are no project with metrics: the score is 0 for all projects;
  - If there are projects with metrics: the projects without metrics are filtered out and the scores of the ones with metrics are computed and compared.

  This step is a **ranking** step.
  - The score formula for a OpenStack project without metrics is defined as follows:

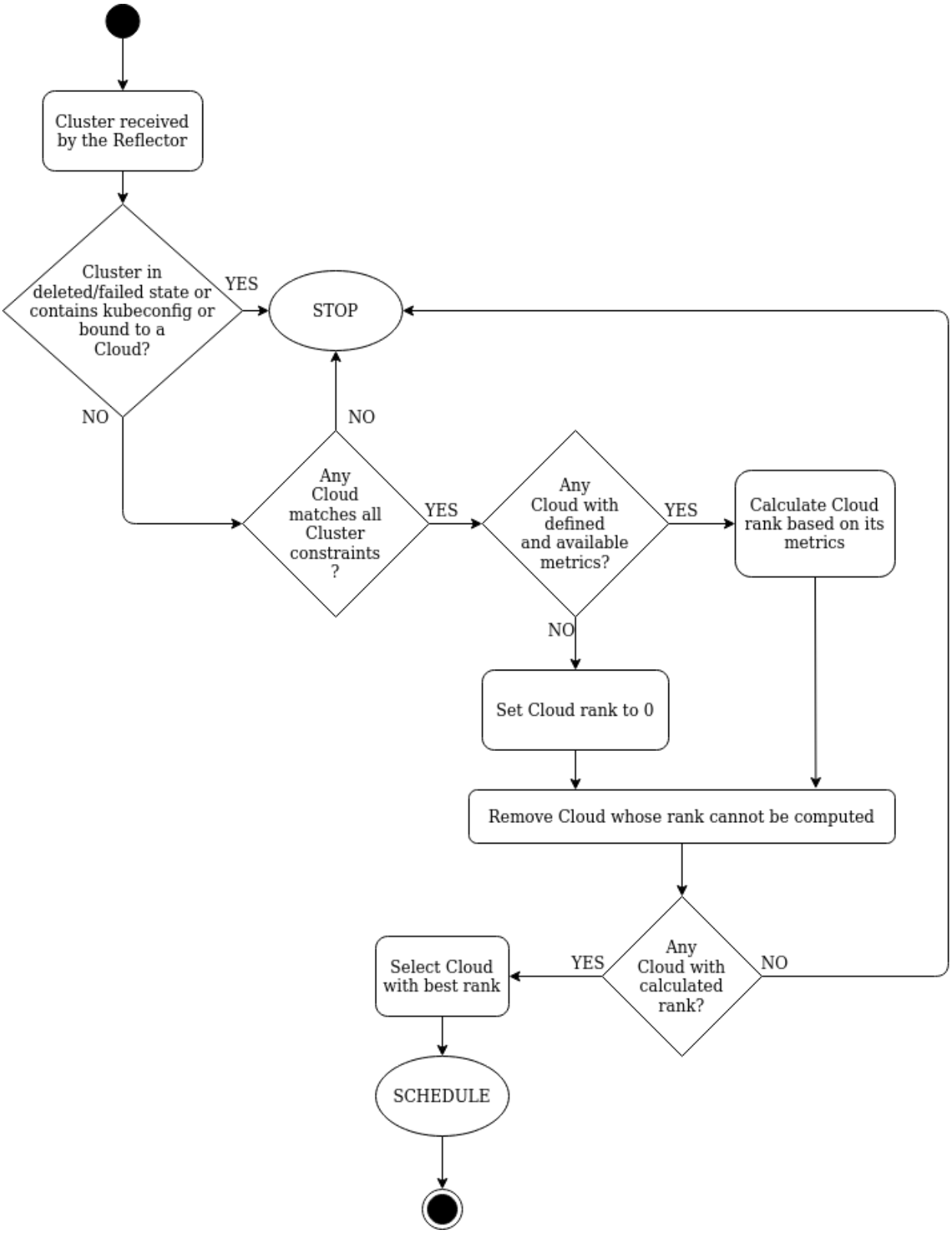
$$score = 0$$

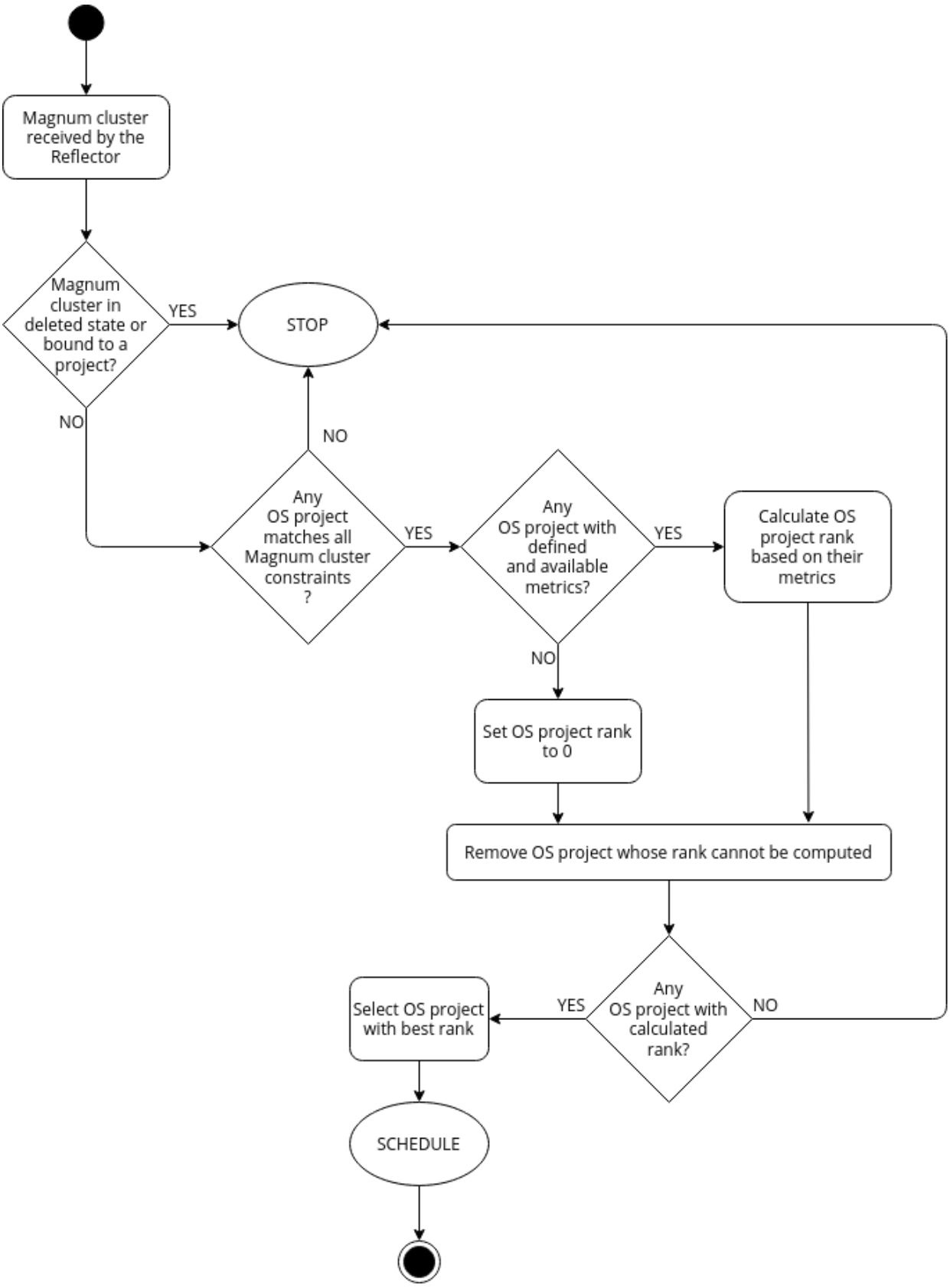
  - The score formula for a OpenStack project with *n* metrics is defined as follows:

$$score = \frac{\sum_{i=1}^{n} metric_{value_i} \cdot metric_{weight_i}}{\sum_{i=1}^{n} metric_{weight_i}}$$

- The Magnum cluster is scheduled to the OpenStack project with the highest score. If several OpenStack projects have the same score, one of them is chosen randomly.

The following figure gives an overview about the Magnum cluster handler of Krake scheduler. "OS project" means "OpenStack project resource" on the figure.


Cluster received by the Reflector
Cluster in deleted/failed state or contains kubeconfig or bound to a Cloud?
YES
STOP
NO
NO
Any Cloud matches all Cluster constraints ?
YES
Any Cloud with defined and available metrics?
YES
Calculate Cloud rank based on its metrics
NO
Set Cloud rank to 0
Remove Cloud whose rank cannot be computed
Any Cloud with calculated rank?
YES
NO
Select Cloud with best rank
SCHEDULE


Magnum cluster received by the Reflector
Magnum cluster in deleted state or bound to a project?
YES
STOP
NO
NO
Any OS project matches all Magnum cluster constraints ?
YES
Any OS project with defined and available metrics?
YES
Calculate OS project rank based on their metrics
NO
Set OS project rank to 0
Remove OS project whose rank cannot be computed
Any OS project with calculated rank?
YES
NO
Select OS project with best rank
SCHEDULE

4.5.4 Metrics and Metrics Providers

Overview

Warning: Due to stability and development issues on the side of Magnum, this feature isn't actively developed anymore.

This section describes the metrics and their providers used in the Krake scheduling algorithm.

The Krake scheduler filters backends based on defined backend metrics. The appropriate metrics definition can prioritize the backend as a potential destination for a given resource.

Krake provides two kinds of Metrics and MetricsProviders. `GlobalMetric` as well as the `GlobalMetricsProvider` can be used throughout the entire Krake infrastructure by all users, apps and clusters. In contrast, the `Metric` and `MetricsProvider` object are bound to a namespace (hence why they're called 'namespaced') and can only be used in their respective context. In most of the documentation chapters, only `GlobalMetrics` are talked about, but namespaced `Metrics` can also be used to follow these sections.

The metrics for the Kubernetes clusters, Magnum clusters and OpenStack projects resources are defined by the `-m` or `--metric` option in the rok CLI, see *Rok documentation*. Multiple metrics can be specified for one resource with the following syntax: <*name*> <*weight*>.

Examples:

```
# Kubernetes clusters:
rok kube cluster create <kubeconfig> --metric heat_demand_zone_1 0.45

# Magnum clusters:
rok os cluster create <cluster_name> --metric heat_demand_zone_1 54

# OpenStack projects:
rok os project create --user-id $OS_USER_ID --template $TEMPLATE_ID my-project --
↪metric heat_demand_zone_1 3
```

By design, the general Krake metric resource (called `GlobalMetric`) is a core api object, that contains its value normalization interval (min, max) and metrics provider name, from which the metric current value should be requested. For the moment, Krake supports the following types of metrics providers:

- **Prometheus** metrics provider, which can be used to fetch the current value of a metric from a Prometheus server;
- **Kafka** metrics provider, which can be used to fetch the current value of a metric from a KSQL database;
- **Static** metrics provider, which returns always the same value when a metric is fetched. Different metrics can be configured to be given by a Static provider, each with their respective value. The static provider was mostly designed for testing purposes.

The metrics provider is defined as a core api resource (called `GlobalMetricsProvider`) that stores the access information for the case of a Prometheus metrics provider, or the metrics values for the case of a Static metrics provider.

Example

```
api: core
kind: GlobalMetric
metadata:
```

(continues on next page)

(continued from previous page)

```
  name: heat_demand_zone_1  # name as stored in Krake API (for management purposes)
spec:
  max: 5.0
  min: 0.0
  provider:
    metric: heat_demand_zone_1  # name on the provider
    name: <metrics provider name> # for instance prometheus or static_provider

---
# Prometheus metrics provider
api: core
kind: GlobalMetricsProvider
metadata:
  name: prometheus_provider
spec:
  type: prometheus  # specify here the type of metrics provider
  prometheus:
    url: http://localhost:9090

---
# Kafka metrics provider
api: core
kind: GlobalMetricsProvider
metadata:
  name: kafka_provider
spec:
  type: kafka
  kafka:
    comparison_column: my_comp_col  # Name of the column where the metrics names are␣
↪stored
    table: my_table  # Name of the table in which the metrics are stored
    url: http://localhost:8080
    value_column: my_value_col  # Name of the column where the metrics values are␣
↪stored

---
# Static metrics provider
api: core
kind: GlobalMetricsProvider
metadata:
  name: static_provider
spec:
  type: static  # specify here the type of metrics provider
  static:
    metrics:
      heat_demand_zone_1: 0.9
      electricity_cost_1: 0.1
```

In the example above, all metrics providers could be used to fetch the `heat_demand_zone_1` metric. By specifying a name in `spec.provider.name` of the `GlobalMetric` resource, the value would be fetched from a different provider:

- `prometheus_provider` for the Prometheus provider;
- `kafka_provider` for the Kafka provider;
- `static_provider` for the Static provider (and the metric would always have the value `0.9`).

Note: A metric contains two "names", but they can be different. `metadata.name` is the name of the GlobalMetric resource as stored by the Krake API. In the database, there can not be two resources of the same kind with the exact same name.

However (if we take for instance the case of Prometheus), two metrics, taken from two different Prometheus servers could have the exact same name. This name is given by `spec.provider.metric`.

So two Krake *GlobalMetric's resources could be called ''latency_from_A'* and `latency_from_B` in the database, but their name could be `latency` in both Prometheus servers.

The Krake metrics and metrics providers definitions can also be added directly to the Krake etcd database using the script *krake_bootstrap_db*, instead of using the API, see *Bootstrapping*.

4.5.5 Constraints

This section describes the resource constraints definition used in the Krake scheduling algorithm.

The Krake scheduler filters appropriate backends based on defined resource constraints. A backend can be accepted by the scheduler as a potential destination for a given resource only if it matches all defined resource constraints.

The Krake scheduler supports the following resource constraints:

- Label constraints
- Metric constraints
- Custom resources constraints

The Krake users are allowed to define these restrictions for the scheduling algorithm of Krake.

The following sections describe the supported constraints of the Krake scheduler in more detail.

Label constraints

Krake allows the user to define a label constraint and to restrict the deployment of resources only to backends that matches **all** defined labels. Based on the resource, Krake supports the following label constraints:

- The Cluster label constraints for the Application resource
- The Cloud label constraints for the Cluster resource
- The OpenStack project label constraints for the Magnum Cluster resource (Warning: Due to stability and development issues on the side of Magnum, this feature isn't actively developed anymore.)

A simple language for expressing label constraints is used. The following operations can be expressed:

equality The value of a label must be equal to a specific value:

```
<label> is <value>
<label> = <value>
<label> == <value>
```

non-equality The value of a label must not be equal to a specific value:

```
<label> is not <value>
<label> != <value>
```

inclusion The value of a label must be inside a set of values:

```
<label> in (<value>, <value>, ...)
```

exclusion The value of a label must not be inside a set of values:

```
<label> not in (<value>, <value>, ...)
```

The Cluster label constraints for the Application and Cluster resources are defined by `-L` (or `--cluster-label-constraint`, `--cloud-label-constraint`) option in the rok CLI, see *Rok documentation*. The constraints can be specified multiple times with the syntax: *<label> expression <value>*.

Examples:

```
# Application
rok kube app create <application_name> -f <path_to_manifest> -L 'location is DE'

# Cluster:
rok kube cluster create <cluster_name> -f <path_to_tosca> -L 'location is DE' ...
```

Metric constraints

Krake allows the user to define a metric constraint and to restrict the deployment of resources only to backends that matches the metric constraint. Based on the resource, Krake supports the following metric constraints:

- The Cluster metric constraints for the Application resource
- The Cloud metric constraints for the Cluster resource

A simple language for expressing metric constraints is used. The following operations can be expressed:

equality The value of a label must be equal to a specific value:

```
<metric> is <value>
<metric> = <value>
<metric> == <value>
```

non-equality The value of a metric must not be equal to a specific value:

```
<metric> is not <value>
<metric> != <value>
```

greater than The value of a metric must be greater than a specific value:

```
<metric> greater than <value>
<metric> gt <value>
<metric> > <value>
```

greater than or equal The value of a metric must be greater or equal than a specific value:

```
<metric> greater than or equal <value>
<metric> gte <value>
<metric> >= <value>
<metric> => <value>
```

less than The value of a metric must be less than a specific value:

```
<metric> less than <value>
<metric> lt <value>
<metric> < <value>
```

less than or equal The value of a metric must be less or equal than a specific value:

```
<metric> less than or equal <value>
<metric> lte <value>
<metric> <= <value>
<metric> =< <value>
```

The metric label constraints for the Application and Cluster resources are defined by `-M` (or `--cluster-metric-constraint`, `--cloud-metric-constraint`) option in the rok CLI, see *Rok documentation*. The constraints can be specified multiple times with the syntax: *<metric> expression <value>*.

Examples:

```
# Application
rok kube app create <application_name> -f <path_to_manifest> -M 'load = 5'

# Cluster
rok kube cluster create <cluster_name> -f <path_to_tosca> -M 'load = 5' ...
```

Custom resources:

Krake allows the user to deploy an application that uses Kubernetes Custom Resources (CR).

The user can define which CRs are available on his cluster. A CR is defined by the Custom Resource Definition (CRD) and Krake uses this CRD name with the format `<plural>.<group>` as a marker.

The supported CRD names are defined by `-R` or `--custom-resource` option in rok CLI. See also *Rok documentation*.

Example:

```
rok kube cluster create <kubeconfig> --custom-resource <plural>.<group>
```

Applications that are based on a CR have to be explicitly labeled with a cluster resource constraint. This is used in the Krake scheduling algorithm to select an appropriate cluster where the CR is supported.

Cluster resource constraints are defined by a CRD name with the format `<plural>.<group>` using `-R` or `--cluster-resource-constraint` option in rok CLI. See also *Rok documentation*.

Example:

```
rok kube app create <application_name> -f <path_to_manifest> --cluster-resource-
↪constraint <plural>.<group>
```

4.6 Application hooks

This section describes Application hooks which are registered and called by the Hook Dispatcher in kubernetes application controller module.

4.6.1 Complete

The application *complete* hook gives the ability to signals job completion.

The Krake Kubernetes controller calls the application *complete* hook before the deployment of the application on a Kubernetes cluster. The hook is disabled by default. The user can enable this hook with the `--hook-complete` argument in rok CLI.

See also *Rok documentation*.

The complete hook injects the `KRAKE_COMPLETE_TOKEN` environment variable, which stores the Krake authentication token, and the `KRAKE_COMPLETE_URL` environment variable, which stores the Krake *complete* hook URL for a given application.

By default, this URL is the Krake API endpoint as specified in the Kubernetes Controller configuration. This endpoint may be only internal and thus not accessible by an application that runs on a cluster. Thus, the `external_endpoint` parameter can be leveraged. It specifies an endpoint of the Krake API, which can be accessed by the application. The endpoint is only overridden if the `external_endpoint` parameter is set.

Applications signal the job completion by calling the *complete* hook URL. The token is used for authentication and should be sent in a PUT request body.

4.6.2 Shutdown

The application *shutdown* hook gives the ability to gracefully stop an application before a migration or deletion happens. This in turn allows to save data or bring other important processes to a safe conclusion.

The Krake Kubernetes controller calls the application *shutdown* hook before the deployment of the application on a Kubernetes cluster. The hook is disabled by default. The user can enable this hook with the `--hook-shutdown` argument in rok CLI.

See also *Rok documentation*.

The shutdown hook injects the `KRAKE_SHUTDOWN_TOKEN` and the `KRAKE_SHUTDOWN_URL` environment variables, which respectively store the Krake authentication token and the Krake *shutdown* hook URL for a given application.

By default, this URL is the Krake API endpoint as specified in the Kubernetes Controller configuration. This endpoint may be only internal and thus not accessible by an application that runs on a cluster. Thus, the `external_endpoint` parameter can be leveraged. It specifies an endpoint of the Krake API, which can be accessed by the application. The endpoint is only overridden if the `external_endpoint` parameter is set.

If the application should be migrated or deleted, Krake calls the `shutdown` services URL, which is set via the manifest file of the application. The integrated service gracefully shuts down the application, preferably via SIGTERM call, but the exact implementation is up to the individual developer. After the shutdown process is complete, the service sends a completion signal to the *shutdown* hook endpoint of the specific application on the Krake API. The previously set token is used for authentication and should be sent in a PUT request body. This requirement prevents the malicious or unintentional deletion of an application. The workflow of this process can be seen in the following figure:

The shutdown hook was developed especially to enable stateful applications. Since these services generate data or are in specific states, it was difficult to migrate or even delete these applications without disrupting their workflow. The shutdown hook enables these normal Krake features for these applications by allowing saving of the current state. But be aware, that Krake doesn't implement a specific graceful shutdown for these applications and merely gives them a possibility to be informed about the intentions of Krake.

4.6.3 TLS

If TLS is enabled on the Krake API, both hooks need to be authenticated with some certificates signed directly or indirectly by the Krake CA. For that purpose, the hooks inject a Kubernetes ConfigMap for different files and mounts it in a volume:

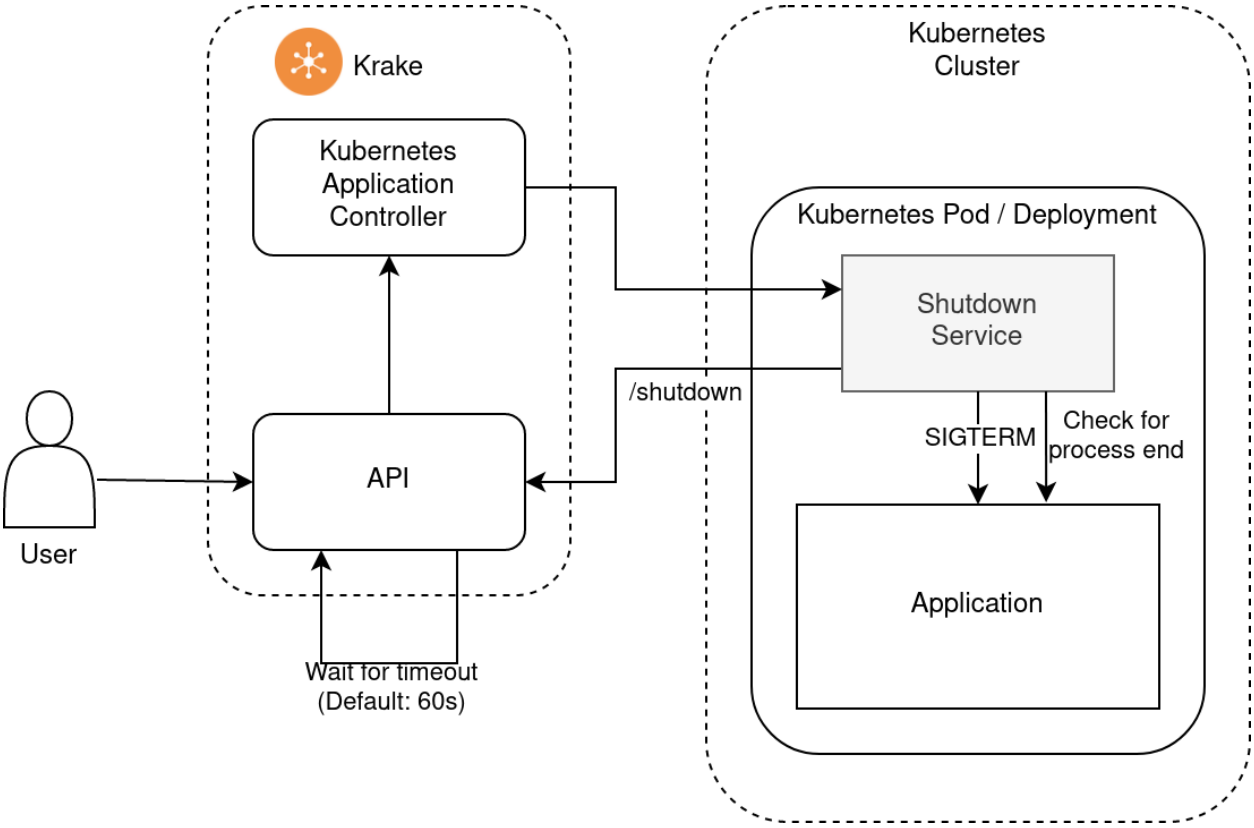


Fig. 3: Shutdown hook workflow in Krake

ca-bundle.pem It contains the CA certificate of Krake, and the hook certificate that was used to sign the certificate specific to the Application.

cert.pem The certificate signed by the hook. It is generated automatically for each Application. Its CN is set to the hooks user defined in the hook configuration, see *Krake configuration*.

key.pem The key of the certificate signed by the hook. It is generated automatically for each Application.

The certificate added are signed by a specific certificate, defined by the `intermediate_src` field in the configuration *Kubernetes application controller*. This certificate needs the following:

- able to sign other certificates;
- hold the right alternative names to accept the Krake endpoint.

The ConfigMap is mounted by default at: `/etc/krake_ca/cert.pem` in the Kubernetes Deployment resources of the Applications.

The name of the environment variables and the directory where the ConfigMap is mounted are defined in the Kubernetes controller configuration file, see *Krake configuration*.

4.6.4 Examples

cURL

Example using *cURL*:

```
$ curl -X PUT -d "{\"token\":\"$KRAKE_COMPLETE_TOKEN\"}" $KRAKE_COMPLETE_URL

# If TLS is enabled on the Krake API
$ curl -X PUT -d "{\"token\":\"$KRAKE_COMPLETE_TOKEN\"}" $KRAKE_COMPLETE_URL \
    --cacert /etc/krake_cert/ca-bundle.pem \
    --cert /etc/krake_cert/cert.pem \
    --key /etc/krake_cert/key.pem
```

By running this command, the Krake API will compare the given token to the one in its database, and if they match, will set the Application to be deleted.

The cURL above may not work with older versions of cURL. You should use versions >= 7.51, otherwise you would get:

```
curl: (35) gnutls_handshake() failed: The TLS connection was non-properly terminated.
```

Python requests

Example using Python's *requests* module:

If TLS is not enabled:

```
import requests
import os

endpoint = os.getenv("KRAKE_COMPLETE_URL")
token = os.getenv("KRAKE_COMPLETE_TOKEN")

requests.put(endpoint, json={"token": token})
```

If TLS is enabled, using the default configuration for the certificate directory:

```
import requests
import os

ca_bundle = "/etc/krake_cert/ca-bundle.pem"
cert_path = "/etc/krake_cert/cert.pem"
key_path = "/etc/krake_cert/key.pem"
cert_and_key = (cert_path, key_path)
endpoint = os.getenv("KRAKE_COMPLETE_URL")
token = os.getenv("KRAKE_COMPLETE_TOKEN")

requests.put(endpoint, verify=ca_bundle, json={"token": token}, cert=cert_and_key)
```

4.7 Kubernetes Application Controller

4.7.1 Reconciliation loop

In the following section, we describe what happens in the Kubernetes Application controller when receiving a resource, and highlight the role of the observer schema.

In this example, the user provides:

- Two resources (one `Song` and one `Artist`) that should be created. This is provided in `spec.manifest`.
- A custom observer schema for the `Song`. This is provided in `spec.observer_schema`

The first resource (`Song`) illustrates the use of a custom observer schema and demonstrates the behavior of list length control. The second resource (`Artist`) highlights the generation of a default observer schema and the special case of mangled resources.

Step 0 (Optional)

If the resource is defined by the TOSCA template file, an URL or a CSAR archive URL, the controller translates the given TOSCA or CSAR file to the Kubernetes manifest file if possible, see *TOSCA*.

The result of translation is stored in `spec.manifest`.

This step is performed by the `ApplicationToscaTranslation` hook.

Step 1

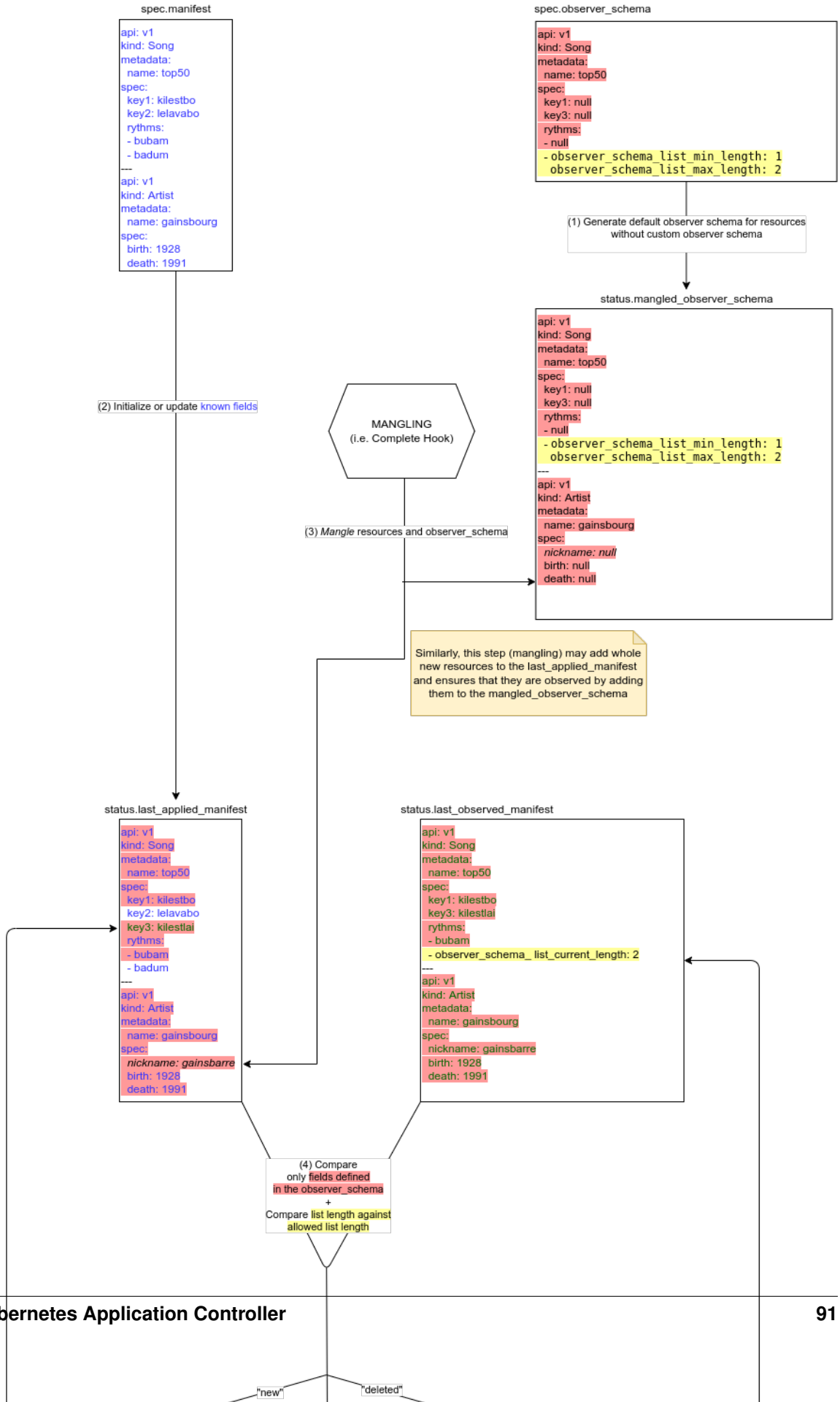
First, the controller generates the default observer schema for resources, where none have been provided. In our example, a default observer schema is created for the `Artist`, while the custom observer schema provided by the user for the `Song` is used as-is.

The result is stored in `status.mangled_observer_schema`.

This step is performed by the `generate_default_observer_schema` function.

Step 2

In this step, the controller initializes - or updates if previously initialized - the `status.last_applied_manifest`. This attribute represents the **desired** state (i.e. which values should be set for which fields).


spec.manifest
api: v1
kind: Song
metadata:
name: top50
spec:
key1: kilestbo
key2: lelavabo
rythms:
- bubam
- badum
---
api: v1
kind: Artist
metadata:
name: gainsbourg
spec:
birth: 1928
death: 1991
spec.observer_schema
api: v1
kind: Song
metadata:
name: top50
spec:
key1: null
key3: null
rythms:
- null
- observer_schema_list_min_length: 1
observer_schema_list_max_length: 2
(1) Generate default observer schema for resources without custom observer schema
status.mangled_observer_schema
api: v1
kind: Song
metadata:
name: top50
spec:
key1: null
key3: null
rythms:
- null
- observer_schema_list_min_length: 1
observer_schema_list_max_length: 2
---
api: v1
kind: Artist
metadata:
name: gainsbourg
spec:
nickname: null
birth: null
death: null
(2) Initialize or update known fields
MANGLING
(i.e. Complete Hook)
(3) Mangle resources and observer_schema
Similarly, this step (mangling) may add whole new resources to the last_applied_manifest and ensures that they are observed by adding them to the mangled_observer_schema
status.last_applied_manifest
api: v1
kind: Song
metadata:
name: top50
spec:
key1: kilestbo
key2: lelavabo
key3: kilestlai
rythms:
- bubam
- badum
---
api: v1
kind: Artist
metadata:
name: gainsbourg
spec:
nickname: gainsbarre
birth: 1928
death: 1991
status.last_observed_manifest
api: v1
kind: Song
metadata:
name: top50
spec:
key1: kilestbo
key3: kilestlai
rythms:
- bubam
- observer_schema_ list_current_length: 2
---
api: v1
kind: Artist
metadata:
name: gainsbourg
spec:
nickname: gainsbarre
birth: 1928
death: 1991
(4) Compare
only fields defined
in the observer_schema
+
Compare list length against
allowed list length
"new"
"deleted"

If empty (i.e. during the first reconciliation of the resource), it is initialized as a copy of `spec.manifest`. The `status.last_applied_manifest` might be augmented at a later step by non-initialized observed fields (see Step 6). As a result, if this field has already been initialized (i.e. during later reconciliation), this step updates the observed fields present in `spec.manifest`.

This the role of the `update_last_applied_manifest_from_spec` function.

In the example above, looking at the `Song` resource:

- `key1` is initialized in `spec.manifest` and is observed.
- `key2` is initialized in `spec.manifest` but is not observed. Its initial value is copied to the `status.last_applied_manifest`, so that the Kubernetes resource can be created using this value. But as it's not observed, its value in `status.last_applied_manifest` will never be updated (see Step 6).
- `key3` is observed but is not set in `spec.manifest`. Its value in `status.last_applied_manifest` is initialized as part of the Step 6 (see below).

Step 3

When an application is mangled, for instance if the Complete Hook has been enabled for the application, some fields or resources are added to `status.last_applied_manifest`. They should also be observed, so there are added to `status.mangled_observer_schema`.

This steps is performed in the `mangle_app` method of the `Complete` class.

In the example above, the `Artist` resource is mangled. The key `spec.nickname` is added to both `spec.last_applied_manifest` and `mangled_observer_schema`.

Step 4

The controller compares the **desired** state (`status.last_applied_manifest`) and the **current** state (represented in `status.last_observed_manifest`). It creates a set of `new`, `updated` and `deleted` resource, to be used in the next step: - `new` resources are present in the **desired** state but not in

the **current** state; they need to be created on the cluster.

- `updated` resources have a different definition in the **desired** and in the **current** state; they need to be updated on the cluster.
- `deleted` resources are not in the **desired** state anymore, but are in the **current** state; they need to be deleted from the cluster.

During the first reconciliation of the application, the **current** state is empty. All resources present in the **desired** state needs to be created.

This steps occurs in `ResourceDelta.calculate()` function.

Note: In order to calculate the "diff" between the desired state and the current state of a resource, the controller: - compares the value of the observed fields only. By definition, the

controller should not act if a non-observed fields value changes.

- checks if the lengths of lists are valid using the list control dictionary.

Step 5

The controller acts on the result of the comparison by either creating, patching, or deleting resources on the cluster. In particular:

- A resource is *created* using the whole `status.last_applied_manifest`. This ensures that all initialized fields (set by the user in `spec.manifest`), are set on the selected cluster, regardless of whether they are observed. In the example above, this is especially the case for `key2` in the `Song`.
- Only the observed fields of a resource are used in order to *patch* that resource.

In other words, the non-observed initialized fields (i.e. set by the user in `spec.manifest`, however not in `spec.observer_schema`): - are used for the creation of the resource. - are not used for patching the resource.

This reflects the fact that if a non-observed fields value changes on the Kubernetes cluster, this update should not be reverted by the Kubernetes Application controller, while providing the user with the ability to set the initial value of a non-observed field.

Step 6

Using the Kubernetes response, the `status.last_applied_manifest` is updated. It is augmented with observed fields which value was not yet known.

In the example above, this is the case of `key3` in the `Song`. It is observed (present in `spec.observer_schema`) but not initialized (not present in `spec.manifest`). Its value in `status.last_applied_manifest` couldn't be initialized during Step 2. Its value is initialized using the Kubernetes response.

This mechanism provides the user with the ability to request a specific field to remain constant, while not providing an initial value for it. It uses the value set initially by the Kubernetes cluster on resource creation.

This task is performed by the hook `update_last_applied_manifest_from_resp`.

Note: Only the observed which are not yet known are added to `status.last_applied_manifest`.

In the unlikely event where a field, which value is already known, has a different value in the Kubernetes response (for instance if `key1` would have a different value in the Kubernetes response), this value is *not* updated in `status.last_applied_manifest`. The user's input prevails in the definition of the **desired** state, represented by `status.last_applied_manifest`.

Note: The `rythms` list possess two elements in the Kubernetes API response. As only the first element is observed, the value of the second element is not saved in `status.last_applied_manifest`.

Step 7

Similarly, the `status.last_observed_manifest` also needs to be updated in order to reflect the **current** state. It holds all observed fields which are present in the Kubernetes response.

This task is performed by the hook `update_last_observed_manifest_from_resp`.

4.8 Kubernetes Application Observer

Krake employs self-healing processes on its resources while running. A reconciliation is done on each resource whose status deviates from its specifications. This can happen if a resource has been modified manually, attacked, or if any anomaly occurred on the actual resource that the Krake resource describes.

4.8.1 Reconciliation

Overview

The reconciliation is the act of bringing the current state of a resource to its desired state. During the course of its life, the real-life pendant of a Krake resource may be updated, and thus differ from the desired state (user-defined). To correct this, Krake performs a reconciliation, and the actual state is "replaced" by the desired state. The Krake Controllers are responsible for actually doing the reconciliation over the resources they manage.

The reconciliation is based on two fields of a resource data structure:

spec The specifications of a resource are stored in this attribute. It corresponds to the **desired** state of this resource.

It has the following properties:

- set and/or updated by the **user**;
- should not be modified by the Krake controllers, but nothing restricts it (should be limited using RBAC, see *Security principles*).

status The **current** status of the resource as seen in the real-world are stored in this attribute.

It has the following properties:

- should not be modified by the user, but nothing restricts it (should be limited using RBAC, see *Security principles*);
- set and/or updated by the **Krake controllers**.

Important: Resources must have a `spec` **AND** a `status` attribute to be reconciled.

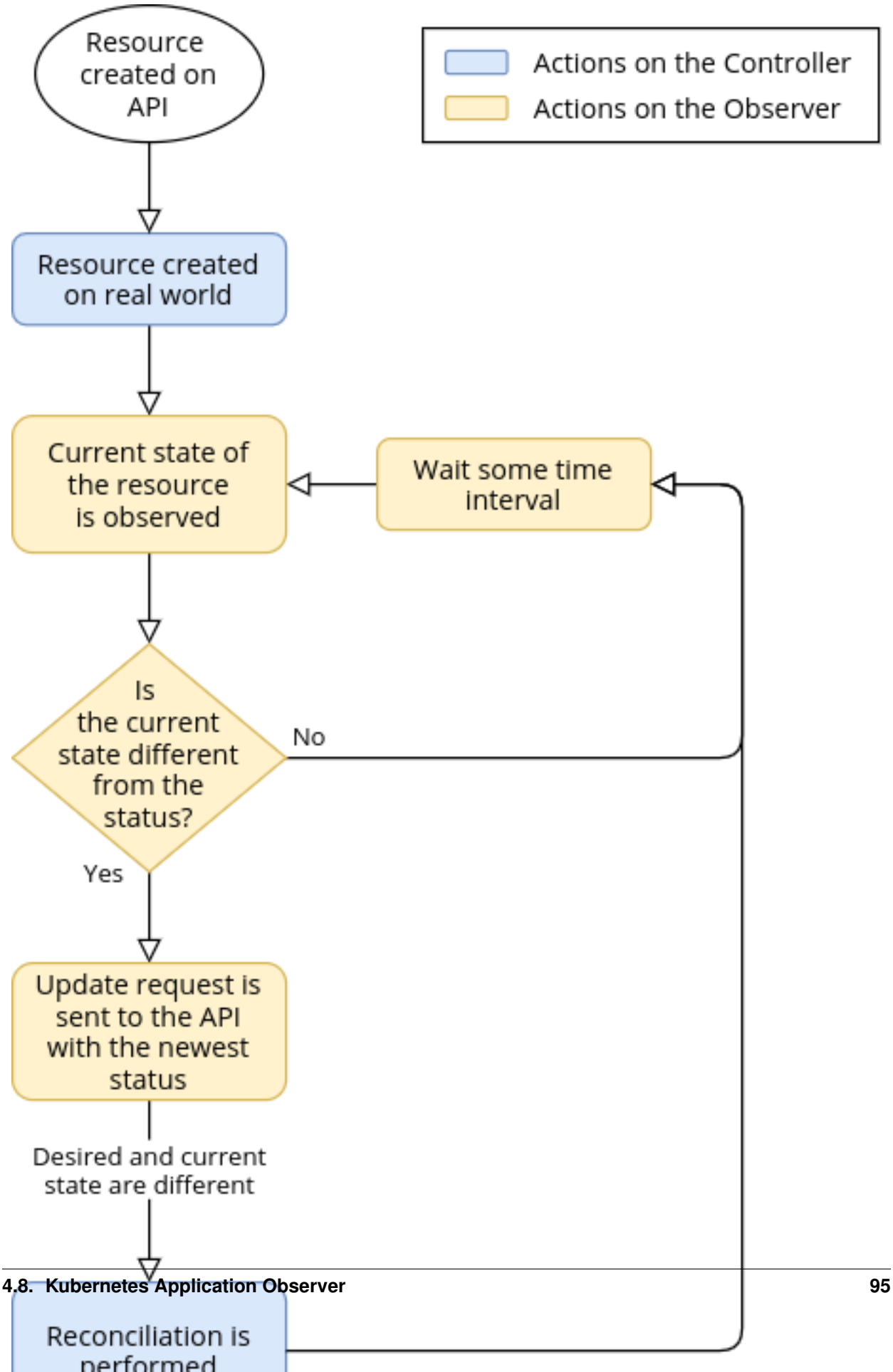
Reconciliation loop

The actual reconciliation is done infinitely, during the so-called reconciliation loop. This loop is not necessarily an in-code loop, and can be more of a conceptual loop between different components.

This workflow in Krake for a specific resource is presented on the following figure:

The workflow is as follows:

1. The resource is created on the API;
2. The actual resource is created by the Controller in the real-world;
3. The Controller responsible for this resources watches, or **observes** its current state in the real-world. This is the role of the **Observer**;
4. This current state is compared to the `status`, stored internally on the Observer;
5. If the actual state is the same as the `status` of the resource, as stored in the Observer, nothing happens. This workflow is started again from step **3** onwards after a defined time period;


Resource created on API
Actions on the Controller
Actions on the Observer
Resource created on real world
Current state of the resource is observed
Wait some time interval
Is the current state different from the status?
No
Yes
Update request is sent to the API with the newest status
Desired and current state are different
Reconciliation is

6. If the actual state is different from the `status` of the resource, it means that the actual resource was modified in the real-world;
7. The Observer notifies the API, by updating the `status` field of the resource;
8. The Controller receives the up-to-date version of the resource, and performs the reconciliation, by applying the desired state on the actual resource;
9. The workflow starts again from step **3**.

Warning: For the moment, Krake only implements reconciliation loop for the Krake `Application` resources of the `Kubernetes` API.

4.8.2 Kubernetes Application Observer

The Krake applications of the `Kubernetes` API have a dedicated KubernetesApplicationObserver. For each application which has some actual resources on a cluster, an observer is created. This KubernetesApplicationObserver watches the status of all Kubernetes resources present in the application specification.

For instance, the `nginx` application has a Kubernetes `Deployment` and a `Service`. If a user changes the image version of the container in the `Deployment` or a label in the `Service`, this will be detected by the Kubernetes application Observer. It will update the `status` of the application and the Kubernetes Controller will observe a deviation with the `spec` and update the actual `Deployment` and `Service` accordingly.

The list of fields which are observed by the Kubernetes application observer can be controlled by specifying a *Custom Observer Schema*.

This observer schema uses the two fields `last_applied_manifest` and `last_observed_manifest`, both of which can be found in `app.status`. `last_applied_manifest` contains the information about the latest applied data, which the application should currently be running on. `last_observed_manifest` on the other hand contains information about the latest observed manifest state of this application. By comparing both datasets, the differences between the desired and observed status can be determined and the corresponding parts can be created, updated or deleted.

The actual workflow of the Kubernetes Application Observer is as follow:

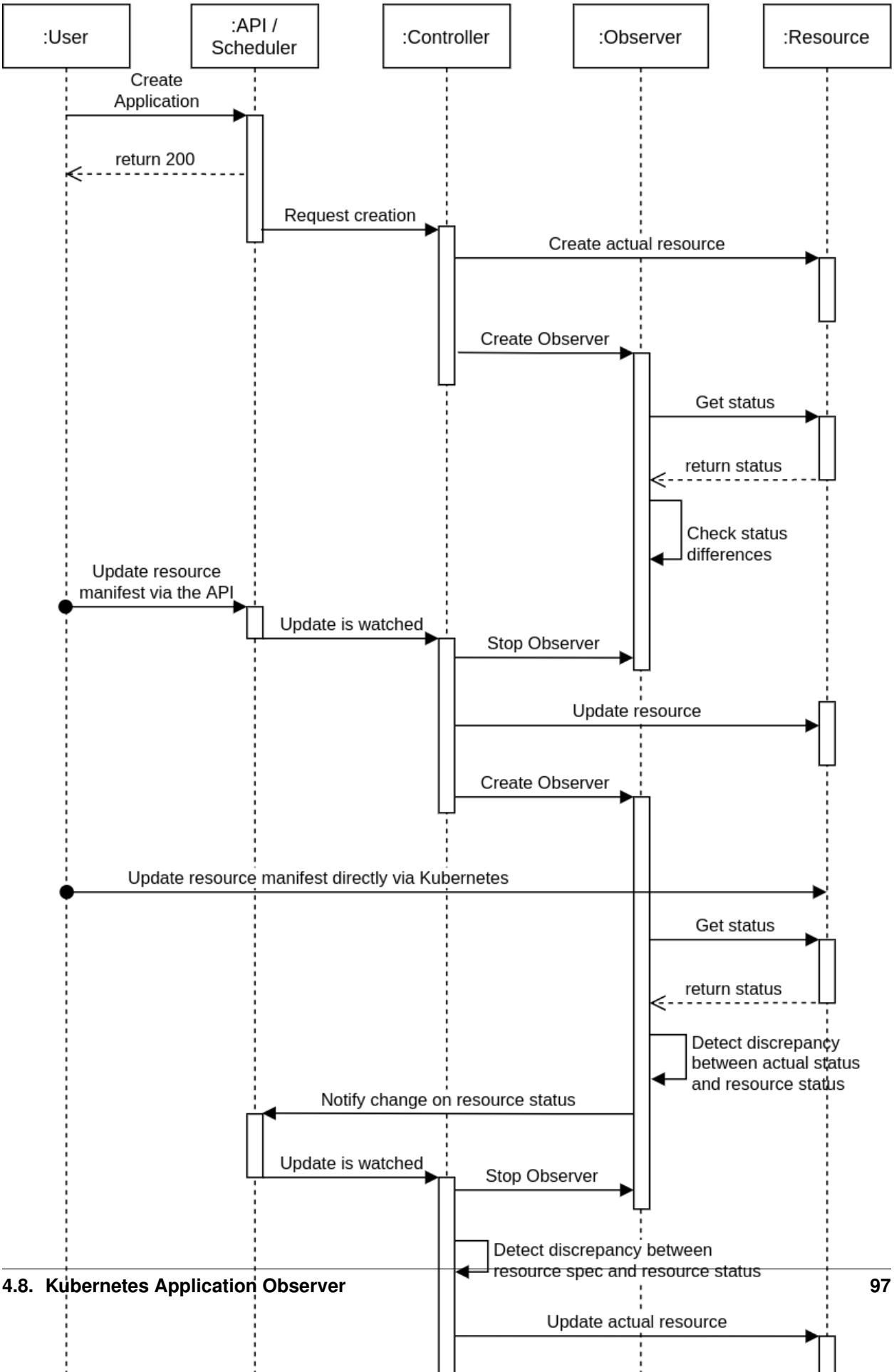
Summary

Creation

After an Application's resources are **created**, a Kubernetes Application Observer is also created for this specific Application.

Update

Before the Kubernetes resources defined in an Application are **updated**, its corresponding Kubernetes Application Observer is stopped. After the update has been performed, a new observer is started, which observes the newest `status` of the Application (the actual Kubernetes resources).


:User
:API / Scheduler
:Controller
:Observer
:Resource
Create Application
return 200
Request creation
Create actual resource
Create Observer
Get status
return status
Check status differences
Update resource manifest via the API
Update is watched
Stop Observer
Update resource
Create Observer
Update resource manifest directly via Kubernetes
Get status
return status
Detect discrepancy between actual status and resource status
Notify change on resource status
Update is watched
Stop Observer
Detect discrepancy between resource spec and resource status
Update actual resource

Deletion

Before the Kubernetes resources of an Application are **deleted**, its corresponding KubernetesApplicationObserver is stopped.

Actions on the API side (summary)

Action	Observer stopped before	Observer started after
Create	No	Yes
Update	Yes	Yes
Delete	Yes	No

On status change

The KubernetesApplicationObserver periodically checks the current state of its Application. The status is read and compared to the `status` field of the Application.

If a Kubernetes resource of the Application changed on its cluster, the KubernetesApplicationObserver sends an update request to the API, to change its `status` field. This field is updated to match what the Observer fetched from the cluster.

Then the Kubernetes Controller starts processing the update normally: a discrepancy is found between the desired state (`spec`) and the current one (`status`). Thus the controller reacts and bring back the current state to match the desired one, by reconciliation. As an update is performed, the observer is stopped before and started after this reconciliation.

After the reconciliation, the `status` field of the Application follows now the desired state. The Kubernetes Application Observer observes this state to check for any divergence.

Warning: If another resource is added manually (not through Krake) to a cluster managed by Krake, Krake will not be aware of it, and no management of this resource will be performed: no migration, self-healing, updates, etc.

4.9 Kubernetes Cluster Controller

The Kubernetes Cluster Controller manages and monitors Kubernetes clusters registered in Krake or created by Krake. To do this for each Kubernetes cluster registered or created by Krake, an observer is created. This observer directly calls the Kubernetes API of the specific cluster and checks on its current state. The Kubernetes cluster controller then updates the internally stored state of the registered or created Kubernetes cluster according to the response from the Kubernetes cluster observer.

The Kubernetes Cluster Controller is launched separately.

Note: Since this is a relatively new implementation, the Kubernetes Cluster Controller will certainly be extended by additional features and functionalities in the future.

For more information on what the Kubernetes Cluster Observer does and what features it offers, see *Kubernetes Cluster Observer*.

For more information about the actual Kubernetes cluster creation by Krake please see *Infrastructure Controller* or visit related user story *Infrastructure providers*.

4.10 Kubernetes Cluster Observer

Krake constantly observes the status of its registered or created clusters while running. For each cluster a separate Kubernetes cluster observer is created. This cluster specific observer calls the Kubernetes API of the real world Kubernetes Cluster periodically. The current status of each cluster is saved in the database of Krake. If a status change of the real world Kubernetes Cluster is detected, Krake updates the saved state of the registered Kubernetes cluster which is stored in its database. Changes in the state of a cluster may be related to Krake being able to connect to the real Kubernetes cluster or not, an unhealthy real world Kubernetes cluster, or metrics providers failing.

4.10.1 Kubernetes Cluster Status Polling

Overview

The `polling` is the act of calling the Kubernetes API of the real world Kubernetes cluster to get its state. During the course of its life, the real-life pendant of a Krake resource may be updated, and thus differ from the desired state (user-defined). To correct this, Krake performs a reconciliation, and the actual state is "replaced" by the desired state. The Krake Controllers are responsible for actually doing the reconciliation over the resources they manage.

Polling

The actual polling is done infinitely, during the so-called polling loop.

This workflow in Krake for a specific resource is presented on the following figure:

The workflow is as follows:

1. The actual real-world cluster is registered or created by Krake;
2. The Kubernetes cluster controller watches, or **observes** the clusters current `status` in the real-world. This is the role of the **KubernetesClusterObserver**. This is done by polling the `status` with a call on the Kubernetes cluster API.
3. This current state is compared to the `status`, stored internally on the Observer;
4. If the actual state is the same as the `status` of the resource, as stored in the Observer, nothing happens. This workflow is started again from step **3** onwards after a defined time period;
5. If the actual state is different from the `status` of the resource, it means that the actual cluster was modified in the real-world;
6. The Observer notifies the Krake API, by updating the `status` field of the cluster;
7. The workflow starts again from step **3**.

4.10.2 States

A Kubernetes cluster watched by it's corresponding KubernetesClusterObserver can have the following observer related states:

- PENDING
- CONNECTING
- ONLINE
- DEGRADED
- OFFLINE

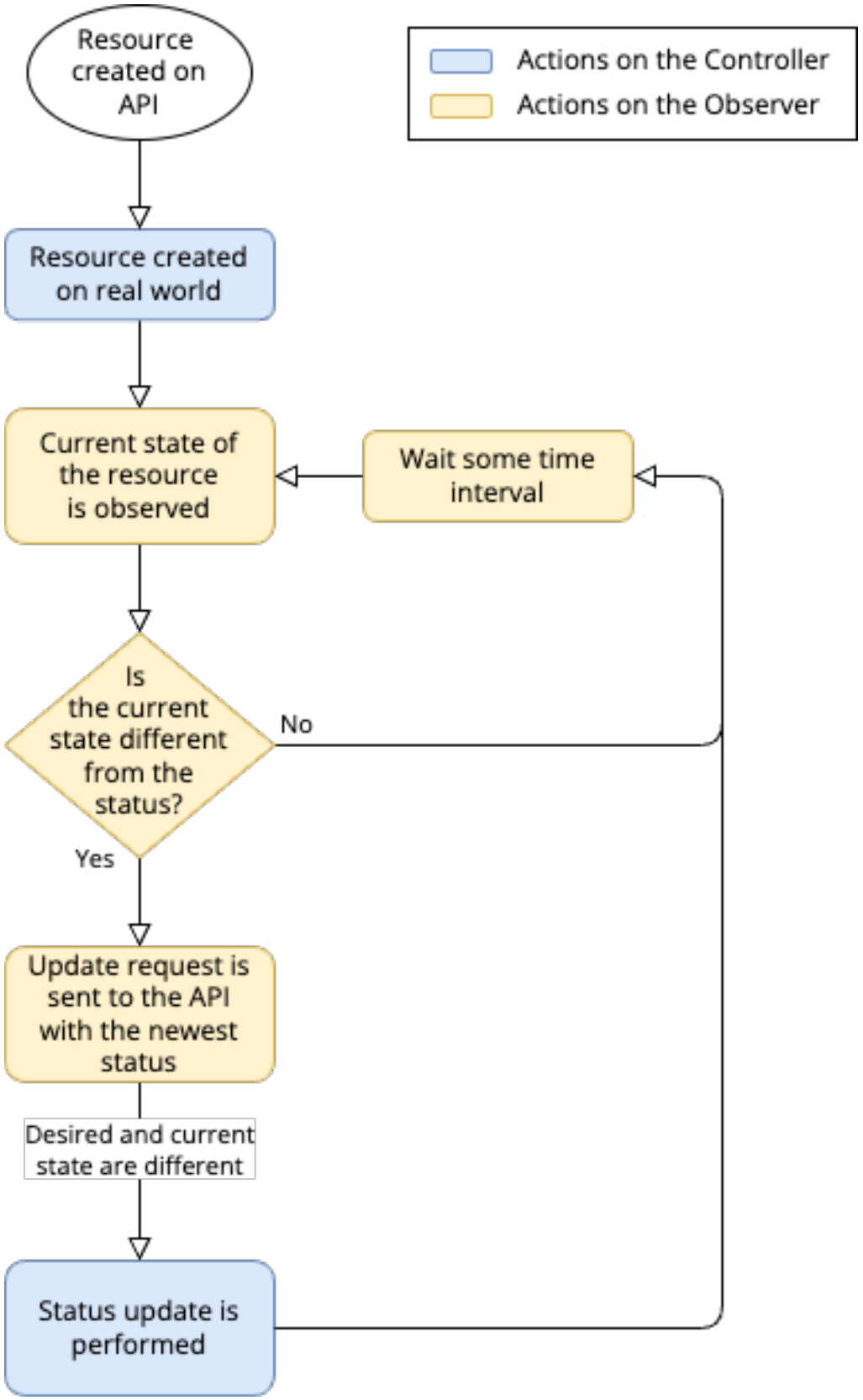


Fig. 6: KubernetesClusterObserver polling loop

- UNHEALTHY
- NOTREADY
- FAILING_METRICS

Note: Refer to the *States* for the infrastructure related cluster states.

PENDING This state is initially set when a Kubernetes cluster is registered in Krake.

CONNECTING It is set by the Kubernetes Cluster Observer if the previous state of a cluster was OFFLINE, but the real cluster is available again. In this case, a reconnection is attempted with a temporary CONNECTING state.

ONLINE If the cluster is reachable in the real world, is in a healthy state and is ready, the status of the cluster in Krake will be ONLINE.

DEGRADED A cluster will be DEGRADED if the handling of the cluster was not successful, but the number of retries is not yet exhausted. This is the intermediate state before being OFFLINE (or back ONLINE). The behaviour is specified with the parameters `backoff` (multiplier added to retry attempts, defaults to 1), `backoff_delay` (number of seconds between retry attempts) and `backoff_limit` (number of retries, defaults to -1(infinite)). So if not changed, the cluster will remain in DEGRADED state until the handling was successful. Otherwise, if the number of retries is exhausted, the cluster will transfer to OFFLINE.

OFFLINE If the real world cluster cannot be reached by polling the Kubernetes cluster API, the status of the cluster in Krake will be OFFLINE. This can happen due to several reasons, e.g. the Kubernetes cluster itself is down, network connectivity issues or incorrect configuration of the used kubeconfig file to register the cluster in Krake.

UNHEALTHY This state is set for clusters in Krake when the Kubernetes API call responds with either PIDPressure, DiskPressure, or MemoryPressure.

NOTREADY This status is displayed when there is an internal problem in the real Kubernetes cluster. When this status is displayed, an investigation of the Kubernetes cluster itself is highly recommended. The reasons for this status vary, for example, the real Kubernetes cluster's kubelet is not working properly or other services have failed to start.

FAILING_METRICS This status is set internally by Krake if a metrics provider is not reachable by Krake and thus metrics cannot be passed correctly into Krake.

4.10.3 Node Health

The cluster observer collects health data of a kubernetes cluster and formats it. Data is divided according to the nodes of the cluster and the different pressure types PID, memory and disk. They represent the problems that a kubernetes node could experience, either missing process ids due to too many process instances, memory overload or non-available disk space. These information can be found by calling:

```
$ rok kube cluster get X
+-----------------------+---------------------+
| ...                   | ...                 |
| nodes                 | 3/3                 |
| nodes_pid_pressure    | 0/3                 |
| nodes_memory_pressure | 0/3                 |
| nodes_disk_pressure   | 0/3                 |
| ...                   | ...                 |
+-----------------------+---------------------+
```

Nodes are shown according to their health, so 3/3 if all nodes are healthy, and the pressure parameters only get filled, if there is a current problem with one (or more) of the nodes.

Summary

Creation

After a Cluster resource was **registered** or successfully **created**, a KubernetesClusterObserver is also created for this specific cluster.

Update

Before the Kubernetes cluster in Krake is **updated**, its corresponding KubernetesClusterObserver is stopped. After the update has been performed, a new observer is started, which observes the newest `status` of the cluster (the actual Kubernetes cluster).

Deletion

Before the Kubernetes cluster is **deleted**, its corresponding KubernetesClusterObserver is stopped.

Actions on the API side (summary)

Action	Observer stopped before	Observer started after
Create	No	Yes
Update	Yes	Yes
Delete	Yes	No

On status change

The KubernetesClusterObserver periodically checks the current state of its cluster. The status is read and compared to the `status` field of the cluster.

If a Kubernetes cluster changed, the KubernetesClusterObserver sends an update request to the API, to change its `status` field. This field is updated to match what the Observer fetched from the cluster.

Then the Kubernetes Cluster Controller starts processing the update normally.

> **Warning:** Currently only `Kubernetes` clusters which have been registered in Krake or created by Krake can be observed.

4.11 Infrastructure Controller

This part of the documentation presents the Infrastructure Controller control plane component, and how the life-cycle management of real-world Kubernetes clusters is handled.

The Infrastructure Controller should process Clusters that are bound (scheduled) to any Cloud or GlobalCloud resource. It should also process Clusters that were deleted and contain an Infrastructure Controller specific deletion finalizer: *infrastructure_resources_deletion*.

Note: Refer to the *Cluster handler* for useful information about cluster scheduling process.

Bound GlobalCloud or Cloud resources correspond to an IaaS cloud deployment (e.g. OpenStack, AWS, etc.) that will be managed by the infrastructure provider backend. Krake currently supports only OpenStack as a GlobalCloud or Cloud backend.

The GlobalCloud or Cloud resource should contain a reference to the GlobalInfrastructureProvider or InfrastructureProvider resource that corresponds to an infrastructure provider backend, that is able to deploy infrastructures (e.g. Virtual machines, Kubernetes clusters, etc.) on IaaS cloud deployments. Krake currently supports only IM (Infrastructure Manager) as an infrastructure provider backend.

Note: The *global* resource (e.g. GlobalInfrastructureProvider, GlobalCloud) is a non-namespaced resource that could be used by any (even namespaced) Krake resource. For example, the GlobalCloud resource could be used by any Cluster which needs to be scheduled to some cloud.

4.11.1 Reconciliation loop

In the following section, we describe what happens in the Infrastructure Controller when receiving a Cluster resource.

Step 1

Infrastructure Controller handles Cluster resources that have been deleted and contain the *infrastructure_resources_deletion* (1). If the above is true, the controller requests the cloud's infrastructure provider for the deletion of the actual cluster counterparts (1a). The controller waits in an infinite loop for the actual cluster deletion (1b). Finally, the controller removes the finalizer from the Cluster resource (1c). This allows the garbage collector controller to remove the Cluster resource from the Krake DB.

Step 2

The Infrastructure Controller handles Cluster resources that are bound (scheduled) to any Cloud or GlobalCloud resource (2). The Cloud or GlobalCloud resource contains cloud API endpoints and access credentials as well as a reference to the infrastructure provider resource through which Krake can manage actual Kubernetes clusters on the bounded cloud.

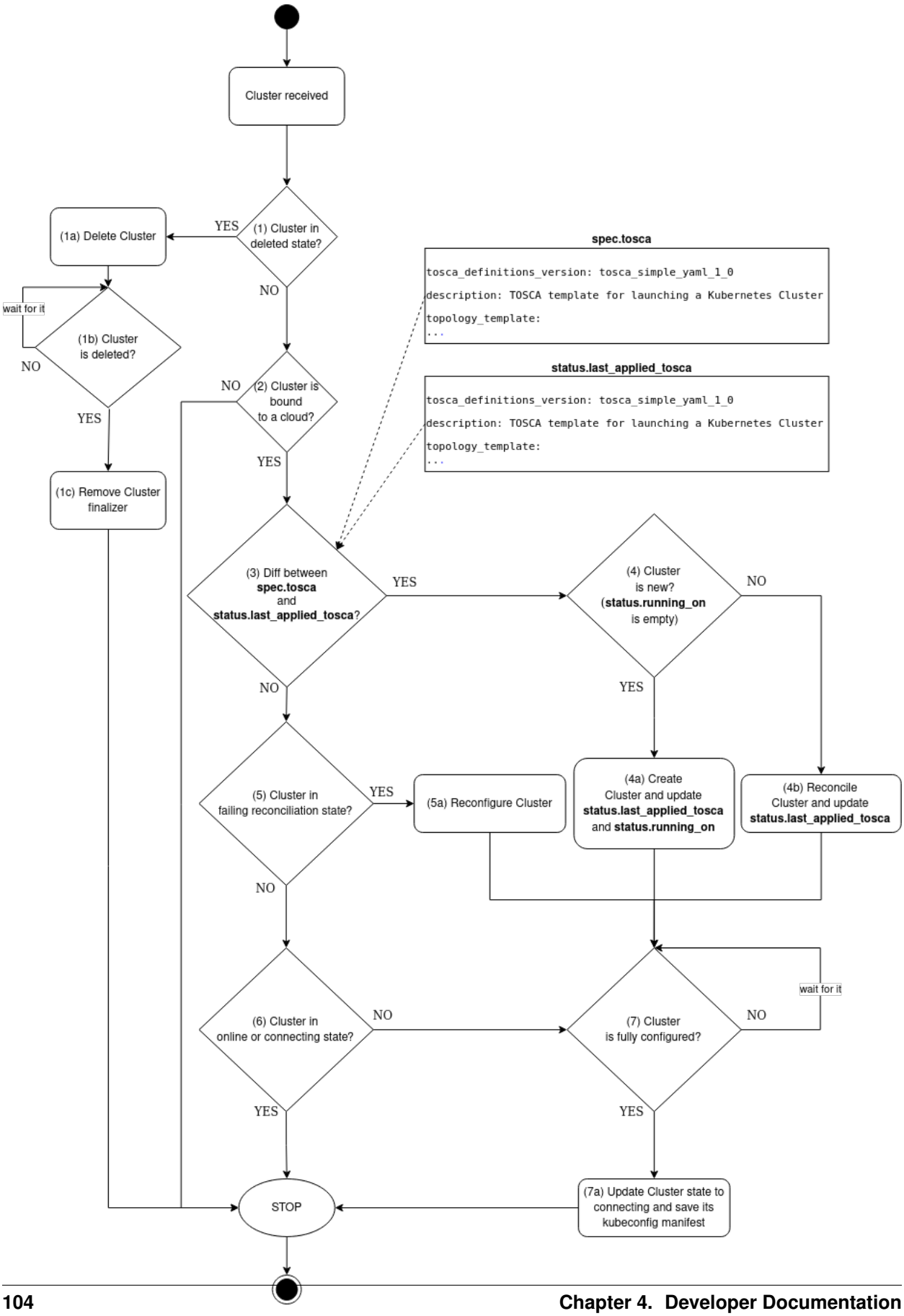
Step 3

If the Cluster is bound (scheduled) to some cloud, the controller recursively looks for all the changes between the desired state (which is represented by the `cluster.spec.tosca` field) and the current state (which is stored in the `cluster.status.last_applied_tosca` field) (3).

Step 4

If there is a difference between the desired and the current state, the controller checks the resource field `cluster.status.running_on` (4).

If it is empty, the resource is considered new, and the controller requests the cloud's infrastructure provider for the creation of the actual cluster counterparts (4a). The TOSCA template stored in `cluster.spec.tosca` represents the desired state and it is applied here. After a successful request for creation, the `cluster.status.last_applied_tosca` field is updated with the copy of the `cluster.spec.tosca` field as well as the


Cluster received
(1) Cluster in deleted state?
YES
(1a) Delete Cluster
NO
wait for it
(1b) Cluster is deleted?
NO
YES
(1c) Remove Cluster finalizer
(2) Cluster is bound to a cloud?
NO
YES
spec.tosca
tosca_definitions_version: tosca_simple_yaml_1_0
description: TOSCA template for launching a Kubernetes Cluster
topology_template:
...
status.last_applied_tosca
tosca_definitions_version: tosca_simple_yaml_1_0
description: TOSCA template for launching a Kubernetes Cluster
topology_template:
...
(3) Diff between spec.tosca and status.last_applied_tosca?
YES
NO
(4) Cluster is new? (status.running_on is empty)
NO
YES
(4a) Create Cluster and update status.last_applied_tosca and status.running_on
(4b) Reconcile Cluster and update status.last_applied_tosca
(5) Cluster in failing reconciliation state?
YES
(5a) Reconfigure Cluster
NO
(6) Cluster in online or connecting state?
NO
YES
(7) Cluster is fully configured?
NO
wait for it
YES
(7a) Update Cluster state to connecting and save its kubeconfig manifest
STOP

`cluster.status.running_on` is updated with the copy of the `cluster.status.scheduled_to` field (scheduled_to field contains the bound cloud resource reference).

If the `cluster.status.running_on` field is not empty, the controller requests the cloud's infrastructure provider for the reconciliation (update) of the actual cluster counterparts (4b). The TOSCA template stored in `cluster.spec.tosca` represents the desired state and it is applied here. After a successful request for reconciliation, the `cluster.status.last_applied_tosca` field is updated with the copy of the `cluster.spec.tosca` field.

Then, the controller waits for the cluster is being successfully configured in the infinite loop (7).

Step 5

If the desired and the current state are in sync, the controller checks whether the Cluster resource state is `FAILING_RECONCILIATION` (5). If so, the controller requests the cloud's infrastructure provider for the reconfiguration of the actual cluster counterparts (5a). This is a "special" call that may or may not be required in case of infrastructure provider failures (e.g. restart). It depends on the underlying infrastructure provider implementation which action should be performed under the hood of the abstract infrastructure controller function *reconfigure*.

Then, the controller waits for the cluster is being successfully configured in the infinite loop (7).

Step 6

The controller finishes the reconciliation if the Cluster resource state is `ONLINE` or `CONNECTING` (6). If it is not the case, the controller waits for the cluster is being successfully configured in the infinite loop (7).

Step 7

The controller waits in an infinite loop for the actual cluster creation/reconciliation/(re)configuration (7). When the actual cluster is fully configured, the controller updates the Cluster state to `CONNECTING` and also saves its kubeconfig manifest to the `cluster.spec.kubeconfig` field. Finally, the controller finishes the reconciliation.

Note: Once the Cluster is configured, has `CONNECTING` state, and contains kubeconfig manifest, the *Kubernetes Cluster Controller* takes over the Cluster and *Kubernetes Cluster Observer* observes its actual status.

4.11.2 States

A Kubernetes Cluster resource managed by the Infrastructure Controller can have the following infrastructure related states:

- PENDING
- CONNECTING
- CREATING
- RECONCILING
- DELETING
- FAILING_RECONCILIATION
- FAILED

Note: Refer to the *States* for the observer related cluster states.

PENDING This state is initially set when a Kubernetes cluster resource is created in Krake.

CONNECTING It is set when the actual Kubernetes cluster has been successfully reconciled.

CREATING It is set when the actual Kubernetes cluster is going to be created.

RECONCILING It is set when the actual Kubernetes cluster is going to be updated.

DELETING It is set when the actual Kubernetes cluster is going to be deleted.

FAILING_RECONCILIATION It is set when the reconciliation process of the actual Kubernetes cluster failed.

FAILED It is set on the global Infrastructure Controller level when an exceptions is raised during the reconciliation process.

Note: Since this is a relatively new implementation, the Infrastructure Controller will certainly be extended by additional features and functionalities in the future, e.g. Infrastructure Observer.

4.12 Garbage Collection

This part of the documentation presents the Garbage Collector component, and how the deletion of resources that others depends on is handled.

4.12.1 Dependency mechanism

In Krake, any resource can depend on any other. In this case, we say the **dependent** depends on the **dependency**. For instance, a Kubernetes Application depends on a Cluster. We also say that the Cluster owns the Application. The Cluster is one of the owners of the Application in this case.

Every resource with metadata holds a list of its owners. However, no resource holds the list of its owned resources. This is similar to the principle of relational database for instance, with the foreign key mechanism.

In the preceding diagram, a `my-app` Application is owned by a cluster (see its list of `owners`) itself belonging to a Magnum cluster. The latter is finally owned by an OpenStack project. The project has no dependency, thus its owner list is empty.

4.12.2 Overview

The Garbage Collector is a Controller of Krake and is, as such, to be started independently from the other components of Krake.

"Marked as deleted" vs "to delete" vs "deleted"

The resources processed during garbage collection have three different states. They use the `"cascade_deletion"` finalizer.

"Marked as deleted" A resource is marked as deleted by the API, when the "delete" action is called on this resource. It means two things for the resource object:

- the `deleted` timestamp of the metadata is set to the current time;

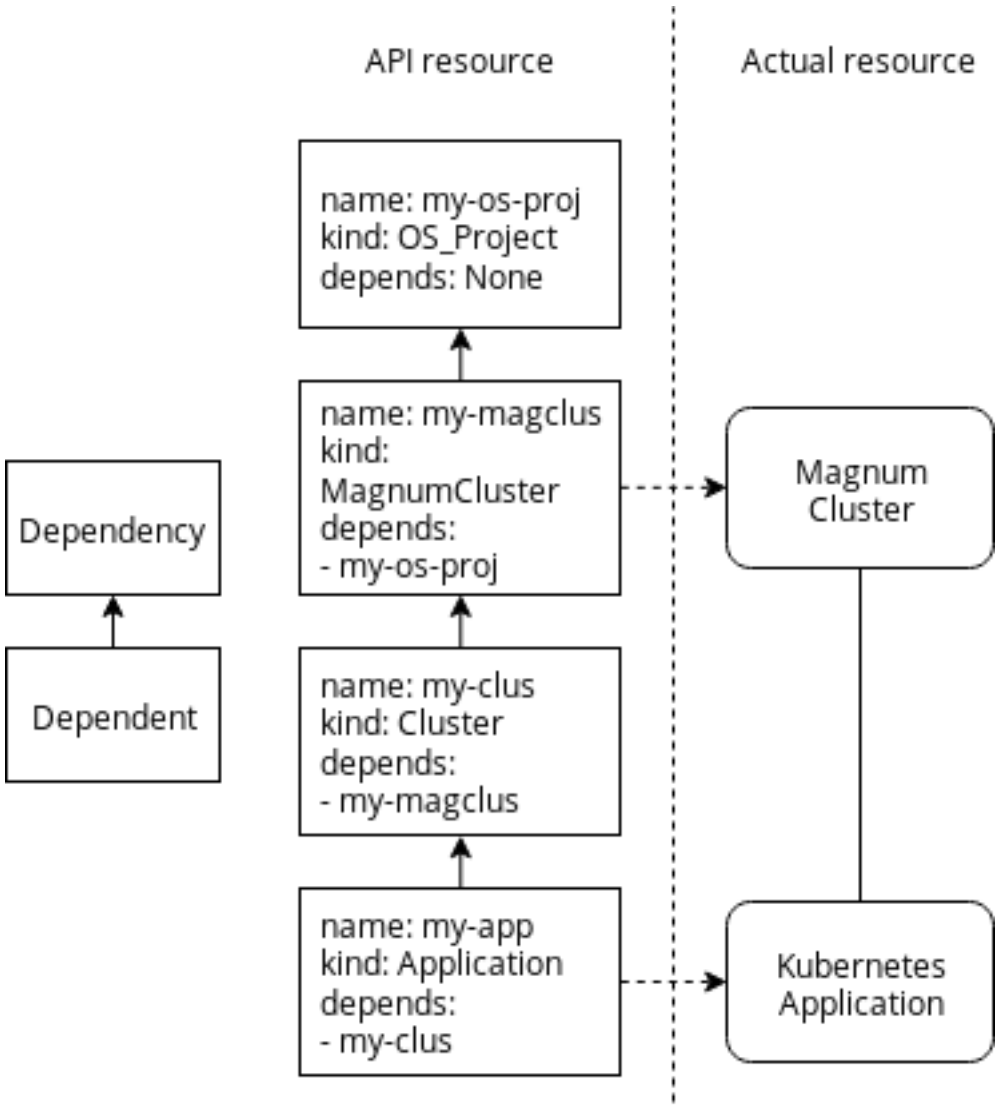


Fig. 7: Dependency relationships in Krake with examples.

- the `"cascade_deletion"` finalizer is added to its list of finalizer.

A resource marked for deletion enters then the garbage collection process.

> **Caution:** This state is irreversible. A resource that enters this state will be processed by the garbage collector, only to be deleted in the end of garbage collection process.

"To delete" A resource is said to be in the "to delete" state if two conditions are met :

- its `deleted` timestamp is set;
- it has no finalizer.

Such a resource can still be transferred by the components. If a resource in this state is received by the API on update, it is deleted.

"Deleted" A deleted resource is completely removed from the database. A last "DELETED" event can be watched on the API when the actual deletion occurs to act on the deletion but the resource itself must be considered erased, and not managed by the API anymore.

Role of the Garbage Collector

The role of the Garbage Collector is to handle resources marked as deleted by the API, but not yet deleted.

When a resource is received, the garbage collector has to:

- update its dependency graph (see *Dependency graph*);
- get the resources that directly depend on it;
- call the API with the "delete" action to let it mark the dependents as deleted;
- if a resource has no dependent, remove the `"cascade_deletion"` finalizer from it, and call the API to update the resource. The resource enters the "to delete" state.

So the role of the Garbage Collector is mostly to get the dependents of a resource, and update them to mark them as deleted. This information is taken from the dependency graph present on the garbage collector, see the *Dependency graph* section.

Role of the API

For the deletion of resources, the garbage collector works tightly with the API, as the garbage collector has no direct access to any resource on the database.

The API is then responsible for:

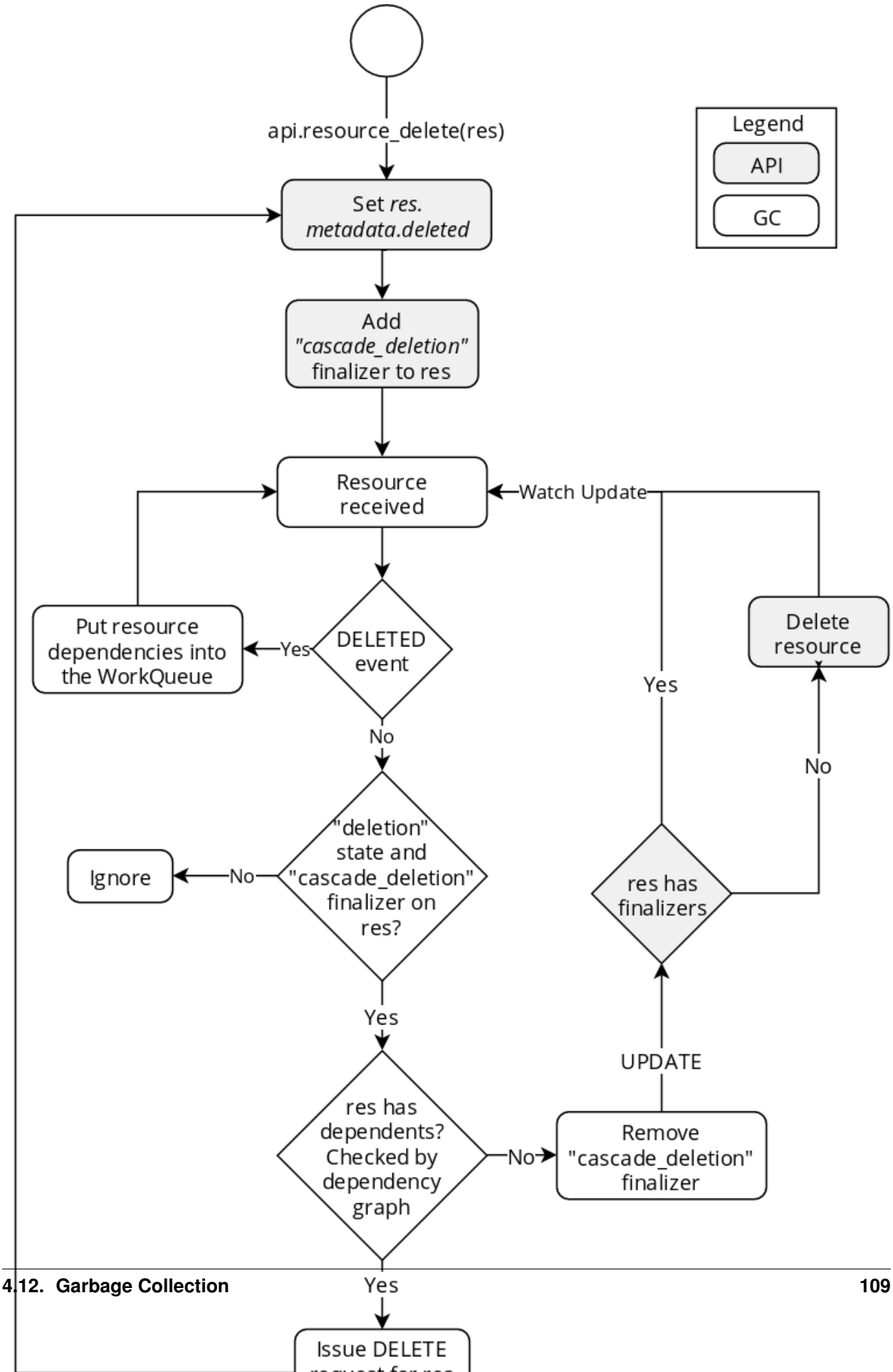
- actually marking the requested resources as deleted;
- completely deleting a resource from the database during an update, if the resource is in a "to delete" state.

So the API is the one that actually modifies and process the stored resources.

4.12.3 Garbage collection workflow

The exact workflow of a resource that the user wants to delete is presented on the previous diagram. Let us take for example an Application **A**, with a cluster **C** as single owner.

1. **A** and **C** were created beforehand, thus they are already present in the dependency graph of the garbage collector;


api.resource_delete(res)
Legend
API
GC
Set res. metadata.deleted
Add "cascade_deletion" finalizer to res
Resource received
Watch Update
Put resource dependencies into the WorkQueue
Yes
DELETED event
No
Delete resource
Yes
No
"deletion" state and "cascade_deletion" finalizer on res?
Ignore
No
res has finalizers
Yes
UPDATE
res has dependents? Checked by dependency graph
No
Remove "cascade_deletion" finalizer
Yes
Issue DELETE

2. the user requests the deletion of the Cluster **C**, for instance with the Rok utility or using `curl`;
3. the request is received by the API. The API marks the cluster **C** as deleted, and an `UPDATE` event is triggered;
4. the garbage collector receives the event. It accepts to handle the cluster, as it is marked for deletion;
5. the list of dependents of **C** is fetched from the dependency graph stored on the garbage collector. The garbage collector issues for each of them a "delete" call to the API. In our case, the Application **A** is the only dependent of **C**;
6. the API receives the call and marks **A** as deleted. **A** is updated, and an `UPDATE` event is triggered;
7. the garbage collector receives the event, and accepts to handle **A**;
8. **A** has no dependent, so its `"cascade_deletion"` finalizer is removed. An "update" request is sent to the API with the new **A**;
9. the API receives the "update" request, with **A** being in the "to delete" state. **A** is deleted from the database. A `DELETED` event is triggered;
10. the garbage collector receives the event. **A** is removed from the dependency graph. The dependencies of **A** are put in the worker queue of the garbage collector to be handled. The owners are collected from the dependency graph. In our case, **C** is added to the worker queue;
11. **C** is handled by the garbage collector a second time. It has no dependent this time, as **A** has been deleted and removed from the dependency graph. Thus, the garbage collector removes the `"cascade_deletion"` finalizer and issues an "update" call to the API for **C**;
12. the API receives the "update" request, with **C** being in the "to delete" state. **C** is deleted from the database. A `DELETED` event is triggered. **C** had no dependency, so the garbage collector does not take any action.

4.12.4 Dependency graph

Description and goal

The dependency graph is an acyclic directed graph stored on the garbage collector as "cache". Its goal is to store the dependency relationships of all resources managed by the API. The graph is updated when starting the garbage collector, while listing resources, or on events triggered by the API. It is only stored in memory, and is re-created each time the garbage collector is started.

The dependency graph allows the garbage collector to access the dependents of any resource. Otherwise, to get the dependents of a resource, the garbage collector would need to request all resources on the database, and check which one of them have the resource to delete as owner. This would mean of course that all resources of the database would be looped through. This is definitely not optimal and is avoided with the dependency graph.

On the nodes, the graph stores the *krake.data.core.ResourceRef* object corresponding to a resource. The edges are directed link from a `ResourceRef` object, to the dependencies of the original object.

krake.data.core.ResourceRef objects are used because they can be keys in dictionaries, whereas normal resources cannot. The reference to the complete resources is still stored in the graph.

Graph workflow

Five actions can be performed on the dependency graph: adding, updating or removing a resource, get the dependents of a resource, or get its dependencies.

Adding a resource: Action performed when the garbage collector lists the resources on startup, or when an "ADDED" event is triggered. The resource is added to the graph as node, along with its dependency relations as edges;

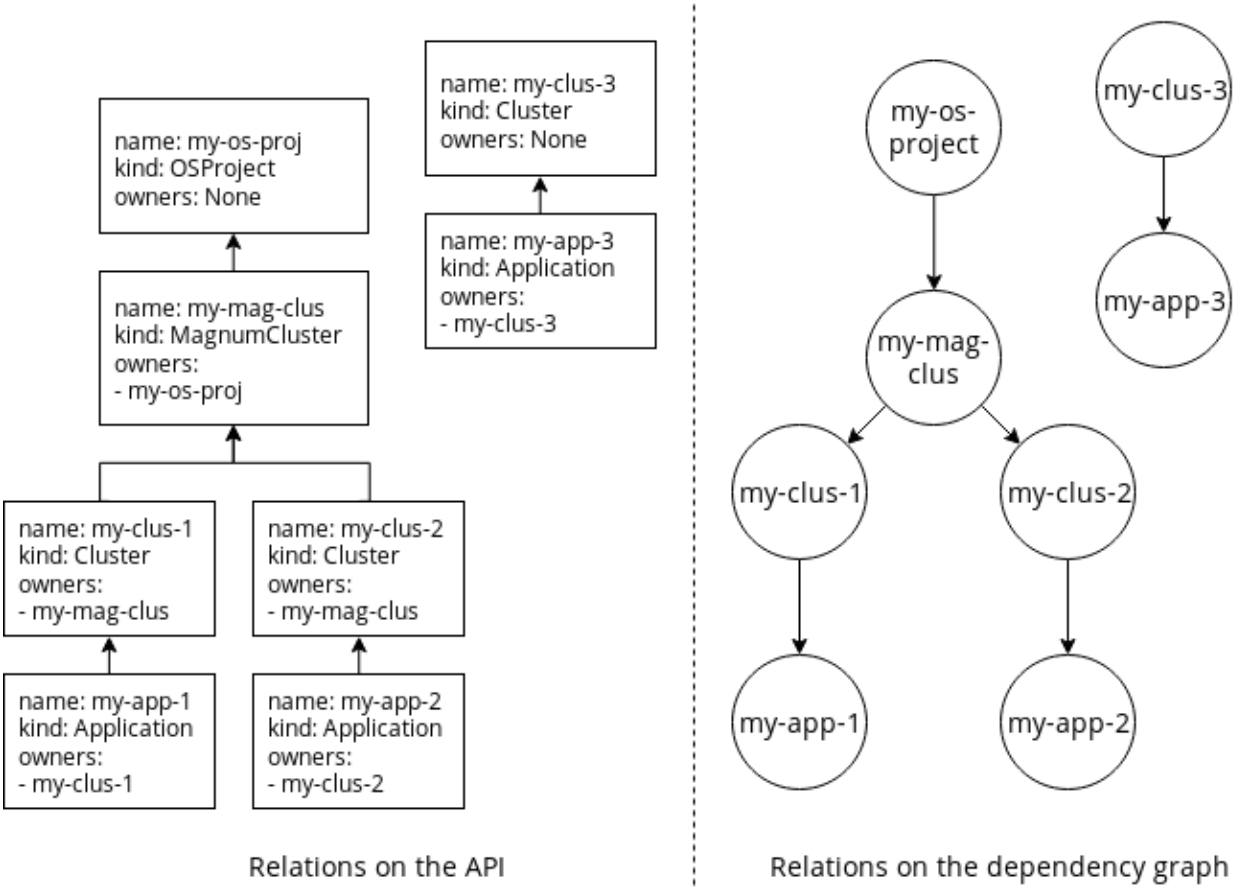
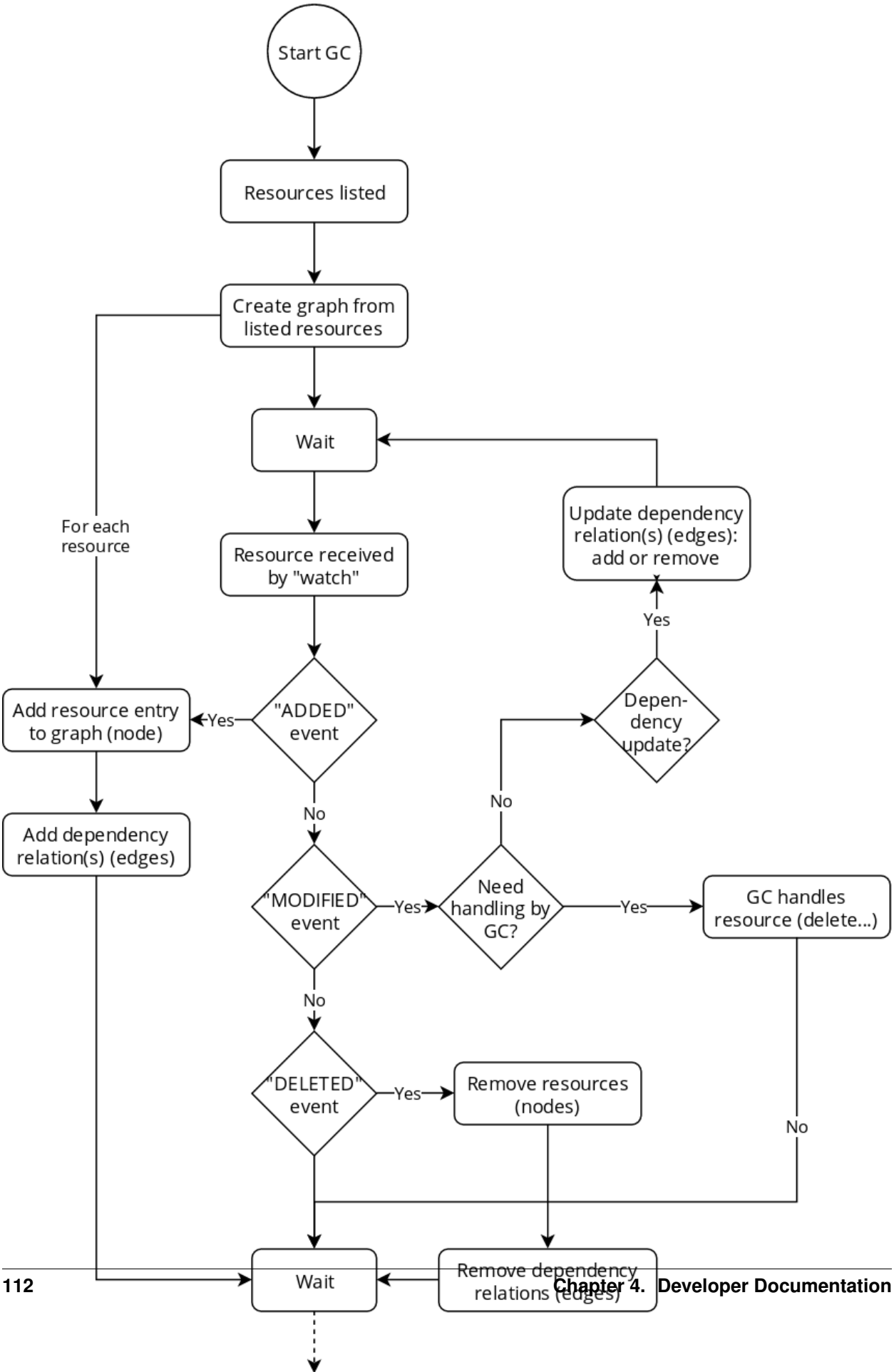


Fig. 9: Comparison example of the dependencies, as represented in the API and on the dependency graph.


Start GC
Resources listed
Create graph from listed resources
Wait
For each resource
Resource received by "watch"
Update dependency relation(s) (edges): add or remove
Yes
Add resource entry to graph (node)
Yes
"ADDED" event
Depen-dency update?
No
Add dependency relation(s) (edges)
No
"MODIFIED" event
Yes
Need handling by GC?
Yes
GC handles resource (delete...)
No
"DELETED" event
Yes
Remove resources (nodes)
No
Wait
Remove dependency relations (edges)

Updating a resource: Action performed when an "UPDATED" event is triggered. If the resource dependency relations were modified, the graph edges are modified. The node corresponding to the resource is modified.

Removing a resource: Action performed when a "DELETED" event is triggered. The resource's corresponding node is removed from the graph, along with the edges bound to it.

Get the dependents of a resource: Action performed by the garbage collector, to know which resource to mark for deletion, without having to reach the API. The nodes on the edges of the resource are listed and returned.

Get the dependencies of a resource: Action performed by the garbage collector, to put the owners of a resource in the worker queue. The owners stored on the resource are returned.

4.13 API Generation

This part of the documentation describes the API generator utility.

4.13.1 Role

The API generator was developed to automatically create the code for:

- the Krake API;
- the client for the Krake API;
- the unit tests for the Krake API;
- the unit tests for the client of the Krake API;
- the API definitions, which are the bases for the generation of the elements above.

Note: Other cases will be added, as the generator was built to be modular.

The Krake API is separated into the different APIs that are managed: `core`, `kubernetes`, `openstack`, and `infrastructure`. Each one of them handles the classic CRUD operations on the different resources managed by the APIs. Having all their code written by hand would not really follow the DRY principle. Previously, the handlers and the client methods were generated dynamically when starting the API. This lead to the code of the API and the client being not very flexible, but mostly, being harder to debug.

As a compromise, the API generator was introduced. It generates the code for any resource of any API in a deterministic way. The code for the API, the client and their respective unit tests are thus more or less "harcoded", as they are not generated on the fly. This has several advantages:

- the code can be easily read, understood, and is accessible easily for debuggers and linters;
- the generation can be nicely integrated with versioning. For instance, when generating new resources or when updating the template of the handlers, the changes can be propagated easily. One only needs to rerun the generator and check the differences.

The API generator should be leveraged in the following cases:

- a new operation on an existing resource is inserted (like the binding for the `Application` resource, or the update of a subresource);
- a new resource is added to an API. All operations to manage it should be handled;
- a whole new API is added. All resources should be managed as well;

4.13.2 Usage

The API generator is a Python module not integrated into the Krake main code. It is present in the Krake repository on `api_generator/`.

Requirements

To install the required packages in your local environment, you can use:

```
$ pip install "krake/[api_generator]"
```

Krake needs to be installed on your local environment as well to be able to use the generator. The previous command also installs Krake.

Commands

The base command for the generator is the following:

```
$ python -m api_generator <command> <parameters>
```

The `<command>` part sets the type of generator which will be used, e.g. the Krake API code, or unit tests for the client. The `<parameters>` are the specific arguments for the chosen generator.

> **Warning:** You need to be in the Krake root directory to use the command.

The command above will display the generated result. To store it into a file, simply redirect the result:

```
$ python -m api_generator <command> <parameters> > <generated_file>
```

4.13.3 Templating

The generated content is based on Jinja templates stored in `api_generator/templates`, but the path can be overwritten, see *Common arguments*. Modifying the templates will modify the generated code, and additional templates can be added for additional operations, unit tests, handlers, etc.

4.13.4 Generated elements

API definitions

The API definitions describe the different operations which can be executed on a type of resource in a specific API. For instance, it would express that the resource `Bar` of API `foo` can be read or listed, but not created, updated or deleted. Additional operations can also be added, for example for bindings, hooks, etc.

To create these definitions automatically, the generation is based on classes defined in the Krake data module. The module inside `krake.data` is imported by the generator, which goes through the module, and filters the classes which will be persistently stored in the database. These classes are considered as being handled by the Krake API, and the operations will only be generated for them.

For each resource (the class handled), the following elements are generated:

- a `Resource` class;
- the singular and plural word for the resource;

- the scope of the resource (namespaced or not);
- basic CRUD operations, plus `List` and `ListAll` (from all namespaces);
- subresource classes inside the `Resource` class for each subresource of the data class (specified by the `"subresource"` metadata of a field being set to `True`.);
- for each subresource, the `Update` operation is generated.

For each operation, the generated definition also describes:

- the HTTP method for the operation;
- the URL path for the operation's endpoint;
- the name of the data class to use for the body of the request to the endpoint;
- the name of the data class that will be used for the body of the response of the Krake API.

For example:

```
$ python -m api_generator api_definition krake.data.kubernetes
```

will generate an API definition file which describes all the resources in the `kubernetes` API of Krake. Among many other elements, a `Status` subresource is added for he `Application` resource.

Regarding the scope, each resource can be either namespaced or non-namespaced. To handle non-namespaced resources, no namespaced should be provided for the API endpoint when calling them. Further, the `List` operation can list all of the elements of the resource, and there is no `ListAll` operation to list all resources of all namespaces (because the instance of the resources are not separated by namespaces).

To specify the scope, use the `--scopes <krake_class_name>=<scope>` argument, once for each resource. For example, for the `foo` API, with resource `Bar` namespaced and `Baz` non-namespaced, the command should be:

```
$ python -m api_generator api_definition krake.data.foo --scopes Baz=NONE
```

After the generation, operations or the attributes of the operation can be changed to restrict or add new operations, change the body of the request or the response, add other subresources, etc.

The existing definitions are stored in the `api_generator/apidefs` directory.

API/client code generation and their unit tests

The generation for the following elements all follow the same procedure:

- code for the Krake API;
- code for the client of the Krake API;
- the unit tests for the Krake API;
- the unit tests for the client of the Krake API.

The four generators leverage the *API definitions* as input. By giving the generator the path to a definition, it will be able to import it and get information from the resources, subresources and their respective operations. This will, in turn, be leveraged for the generation of the code.

```
$ python -m api_generator <command> api_generator.apidefs.foo
```

where the parameter (here `api_generator.apidefs.foo`) is the module path to the API definition used as input, and `<command>` can be:

`api_client`:

The generated output will be code to communicate with the API. For each API, a client class is created, which has a method for each defined operation. These methods take usually a resource as parameter and maybe the name and namespace of a resource. It returns usually the body of the response of the Krake API.

`api_server`:

The generated output will be handlers for the Krake API, to be executed when a request is received. For each operation of each resource, a handler is generated to process the request and prepare the body of the response sent to the client.

`test_client`:

The generated output will be unit tests. They verify the behavior of the client methods generated by the `api_client` command. For each method of the client, several unit tests can be added because of the different behaviors it can have.

`test_server`:

The generated output will be unit tests. They verify the behavior of the handlers generated by the `api_server` command. For each handlers of the API, several unit tests can be added because of the different behaviors it can have.

All these generators share the following common arguments:

- `--operation`
- `--resources`

They can be used to limit respectively the operations and/or the resource that will be handled by the generator for the final output. Can be repeated once for each operation for which the output will be displayed. If one of the option is used, it will only display the mentioned operation or resource. Not using one of them will result in all operations or resources being outputted.

Common arguments

These arguments are common to some generators:

`--no-black`:

to disable the usage of black on the output of the generator before returning it.

`--templates-dir`

to overwrite the templates used for the generation of the code or definitions.

4.14 TOSCA

This section describes TOSCA integration to Krake.

4.14.1 Introduction

TOSCA is an OASIS standard language to describe a topology of cloud-based web services, their components, relationships, and the processes that manage them.

TOSCA uses the concept of service templates to describe services. TOSCA further provides a system of types to describe the possible building blocks for constructing service templates and relationship types to describe possible kinds of relations. It is possible to create custom TOSCA types for building custom TOSCA templates.

Krake allows end-users to orchestrate Kubernetes applications with TOSCA. It is required to use custom TOSCA templates for them. Krake supports Cloud Service Archive (CSAR) files as well. CSAR is a container file using the ZIP file format, that includes all artifacts required to manage the lifecycle of the corresponding cloud application using the TOSCA language.

Note: The TOSCA technical committee has decided that any profile (template base) development should be left to the community. It means, that there are not any "de facto standard" on how to describe e.g. Kubernetes applications with TOSCA. Each orchestrator that supports TOSCA is its own product with its own design paradigms and may have different assumptions and requirements for modeling applications.

4.14.2 TOSCA Template

Krake is able to manage Kubernetes applications that are described by the TOSCA YAML custom template file. Kubernetes application should be described by TOSCA-Simple-Profile-YAML `v1.0` or `v1.2` as Krake only supports those versions.

Krake supports Cloud Service Archives (CSAR) as well. The CSAR should contain TOSCA-Simple-Profile-YAML `v1.0` or `v1.2` and should be in defined format.

Note: Krake uses the tosca-parser library as its underlying TOSCA parser and validator. Currently, tosca-parser supports the TOSCA-Simple-Profile-YAML `v1.0` or `v1.2`, which reflects what is supported by Krake.

Note: The Krake API could process TOSCA templates in two formats. It can receive the TOSCA template as serialized JSON or the API could receive a **URL** that points to some remote location, that provides a TOSCA template. In the case of providing a URL, the underlying tosca-parser library is able to (synchronously) download the TOSCA template from the defined URL and then parse and validate it.

Another prerequisite (besides the TOSCA version) is a TOSCA profile (custom type). Krake supports and can manage only Kubernetes application that is described by the `tosca.nodes.indigo.KubernetesObject` custom type.

The `tosca.nodes.indigo.KubernetesObject` custom type has been defined by the Grycap research group. It could be imported as an external document using the `imports` directive in the template or it can be directly declared as a custom data type within the `data_types` template section.

For import use the following reference to Grycap's custom types:

```
imports:
- ec3_custom_types: https://raw.githubusercontent.com/grycap/ec3/tosca/tosca/custom_
↪types.yaml
```

For direct definition use the following (minimal) data type:

```
data_types:
  tosca.nodes.indigo.KubernetesObject:
    derived_from: tosca.nodes.Root
    properties:
      spec:
        type: string
        description: The YAML description of the K8s object
        required: true
```

The `spec` of `tosca.nodes.indigo.KubernetesObject` custom type should contain Kubernetes manifest as a string. It is possible to applied subset of supported TOSCA functions like:

- get_property
- get_input

The `spec` of the `tosca.nodes.indigo.KubernetesObject` custom type should contain a Kubernetes manifest as a string. It is possible to apply a subset of supported TOSCA functions like:

- get_property
- get_input
- concat

Then, the example of TOSCA template for a single Kubernetes Pod could be designed as follows:

```
tosca_definitions_version: tosca_simple_yaml_1_0

imports:
  - ec3_custom_types: https://raw.githubusercontent.com/grycap/ec3/tosca/tosca/custom_
↪types.yaml

description: TOSCA template for launching an example Pod by Krake

topology_template:
  inputs:
    container_port:
      type: integer
      description: Container port
      default: 80
  node_templates:
    example-pod:
      type: tosca.nodes.indigo.KubernetesObject
      properties:
        spec:
          concat:
            - |-
              apiVersion: v1
              kind: Pod
              metadata:
                name: nginx
              spec:
                containers:
                - name: nginx
                  image: nginx:1.14.2
                  ports:
                  - containerPort:
            - get_input: container_port
```

Let's save the definition above to the `tosca-example.yaml` file.

If you want to expose a created TOSCA template in your localhost, you can use a simple python HTTP server as follows:

```
# TOSCA template will then be exposed on URL: `http://127.0.0.1:8000/tosca-example.
↪yaml`
python3 -m http.server 8000
```

Cloud Service Archive

CSAR should be in a defined format. The specification allows to create CSAR with or without the `TOSCA.meta` file. The `TOSCA.meta` file structure follows the exact same syntax as defined in the TOSCA 1.0 specification. It is required to store this file in the `TOSCA-Metadata` directory. It is also required to include the `Entry-Definitions` keyword pointing to a valid TOSCA definitions YAML file, which should be used by a TOSCA orchestrator as an entrypoint for parsing the contents of the overall CSAR file (the previously created `tosca-example.yaml` file will be used in this example).

Note: The Krake API can process CSAR files **only**, if they're defined as **URL**s. It means, that CSAR should be created and then exposed in some remote location. Then, the underlying tosca-parser library is able to (synchronously) download the CSAR archive from the defined URL and afterwards parse and validate it.

```
# Create TOSCA-Metadata directory
mkdir TOSCA-Metadata
# Create and fill TOSCA.meta file
echo "TOSCA-Meta-File-Version: 1.0" >> TOSCA-Metadata/TOSCA.meta
echo "CSAR-Version: 1.1" >> TOSCA-Metadata/TOSCA.meta
echo "Created-By: Krake" >> TOSCA-Metadata/TOSCA.meta
echo "Entry-Definitions: tosca-example.yaml" >> TOSCA-Metadata/TOSCA.meta
# Create CSAR
zip example.csar -r TOSCA-Metadata/ tosca-example.yaml

# Expose the created CSAR by simple HTTP python server
# CSAR will be then exposed on URL: `http://127.0.0.1:8000/example.csar`
# Expose the created CSAR file with a simple HTTP python server
# CSAR will then be exposed on URL: `http://127.0.0.1:8000/example.csar`
python3 -m http.server 8000
```

4.14.3 TOSCA/CSAR Workflow

The TOSCA template or CSAR archive should be composed on the client side. Then the client sends the request for the creation or update of an application together with the TOSCA template (YAML file or URL) or CSAR URL. The Krake API validates the TOSCA template or CSAR file suffixes depending on the used URL. When the TOSCA template is defined with a YAML file, parsing and validation are performed by Krake API (using the tosca-parser). After validation, the life cycle of the application is the same as a regular one (defined by Kubernetes manifest) except for the translation of the TOSCA template or CSAR archive into a Kubernetes manifest inside of the Kubernetes Application Controller. The controller is responsible for the translation of TOSCA/CSAR to Kubernetes manifests. During this process, the application will in the **TRANSLATING** state.

The workflow of this process can be seen in the following figure:

4.14.4 Examples

Prerequisites

The Krake repository contains a bunch of useful examples. Clone it first with the following commands:

```
git clone https://gitlab.com/rak-n-rok/krake.git
cd krake
```

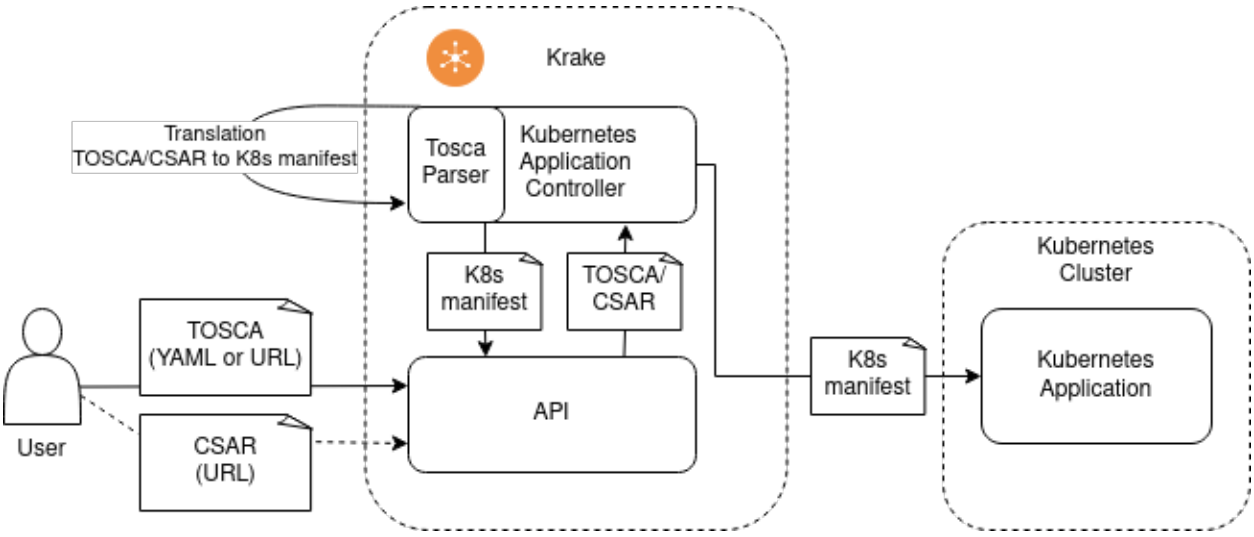


Fig. 11: TOSCA/CSAR workflow in Krake

TOSCA template examples are located in the `rak/functionals` directory. View these TOSCA templates for example:

```
$ cat rak/functionals/echo-demo-tosca.yaml
$ cat rak/functionals/echo-demo-update-tosca.yaml
```

If you want to expose a created TOSCA template via some URL, you can use a simple python HTTP server as follows:

```
cd rak/functionals/
# Expose the TOSCA template examples with a simple HTTP python server
# TOSCA template examples will then be exposed on URLs:
# - `http://127.0.0.1:8000/echo-demo-tosca.yaml`
# - `http://127.0.0.1:8000/echo-demo-update-tosca.yaml`
python3 -m http.server 8000
```

If you are interested in CSAR, use the pre-defined `TOSCA.meta` file and create and expose CSAR archive as follows:

```
cd rak/functionals/
zip echo-demo.csar -r TOSCA-Metadata/ echo-demo-tosca.yaml
# Expose the created CSAR by simple HTTP python server
# CSAR will be then exposed on URL: `http://127.0.0.1:8000/example.csar`
python3 -m http.server 8000
```

Rok

A TOSCA template YAML file should be applied the same way as a Kubernetes manifest file using the rok CLI, see *Rok documentation*.

- Create an application described by a TOSCA template YAML file:

```
rok kube app create --file rak/functionals/echo-demo-tosca.yaml echo-demo
```

- Update an application described by a TOSCA template:

```
rok kube app update --file rak/functionals/echo-demo-update-tosca.yaml echo-demo
```

A TOSCA template URL or CSAR archive URL should be defined after the optional *–url* argument using the rok CLI, see *Rok documentation*.

- Create an application described by a TOSCA template URL:

```
rok kube app create --url http://127.0.0.1:8000/echo-demo-tosca.yaml echo-demo
```

- Update an application described by a TOSCA template URL:

```
rok kube app update --url http://127.0.0.1:8000/echo-demo-update-tosca.yaml echo-demo
```

- Alternatively, create an application described by a CSAR URL:

```
rok kube app create --url http://127.0.0.1:8000/example.csar echo-demo
```

Tip: Krake allows the creation of an application using e.g. a plain Kubernetes manifest and then updating it with a TOSCA or even CSAR file. The same works vice-versa. It means, that the application could be created and then updated by any supported format (Kubernetes manifest, TOSCA, CSAR).

4.15 Krake Reference

This is the code reference for the Krake project.

4.15.1 Module hierarchy

This section presents the modules and sub-modules of the Krake project present in the `krake/` directory.

The tests for Krake are added in the `krake/tests/` directory. The `pytest` module is used to launch all unit tests.

krake The `krake` module itself only contains a few utility functions, as well as functions for reading and validating the environment variables and the configuration provided. However, this module contains several sub-modules presented in the following.

krake.api This module contains the logic needed to start the API as a an `aiohttp` application. It exchanges objects with the various clients defined in *krake.client*. These objects are the ones defined in *krake.data*.

krake.client This module contains all the necessary logic for any kind of client to communicate with the API described in the *krake.api* module.

krake.controller This module contains the base controller and the definition for several controllers. Each one of these controllers is a separate process, that communicates with the API or the database. For this, the controllers use elements provided by the *krake.client* module.

All new controller should be added in this module.

krake.controller.kubernetes.application This sub-module contains the definition of the controller specialized for the Kubernetes application handling.

krake.controller.kubernetes.cluster This sub-module contains the definition of the controller specialized for the Kubernetes cluster handling.

krake.controller.scheduler This sub-module defines the Scheduler controller, responsible for binding the Krake applications and magnum clusters to the specific backends.

krake.controller.gc This sub-module defines the Garbage Collector controller, responsible for handling the dependencies during the deletion of a resource. It marks as deleted all dependents of a resource marked as deleted, thus triggering their deletion.

krake.controller.magnum This sub-module defines the Magnum controller, responsible for managing Magnum cluster resources and creating their respective Kubernetes cluster.

krake.data This module defines all elements used by the API and the controllers. It contains the definition of all these objects, and the logic to allow them to be serialized and deserialized.

4.15.2 Krake

class krake.**ConfigurationOptionMapper**(*config_cls*, *option_fields_mapping=None*)

Bases: object

Handle the creation of command line options for a specific Configuration class. For each attribute of the Configuration, and recursively, an option will be added to set it from the command line. A mapping between the option name and the hierarchical list of fields is created. Nested options keep the upper layers as prefixes, which are separated by a "-" character.

For instance, the following classes:

```
class SpaceShipConfiguration(Serializable):
    name: str
    propulsion: PropulsionConfiguration

class PropulsionConfiguration(Serializable):
    power: int
    engine_type: TypeConfiguration

class TypeConfiguration(Serializable):
    name: str
```

Will be transformed into the following options:

```
--name str
--propulsion-power int
--propulsion-engine-type-name: str
```

And the option-fields mapping will be:

```
{
    "name": [Field(name="name", ...)],
    "propulsion-power": [
        Field(name="propulsion", ...), Field(name="power", ...)
    ],
    "propulsion-engine-type-name": [
        Field(name="propulsion", ...),
        Field(name="engine_type", ...),
        Field(name="name", ...),
    ],
}
```

Then, from parsed arguments, the default value of an element of configuration are replaced by the elements set by the user through the parser, using this mapping.

The mapping of the option name to the list of fields is necessary here because a configuration element called `"lorem-ipsum"` with a `"dolor-sit-amet"` element will be transformed into a `"--lorem-ipsum-dolor-sit-amet"` option. It will then be parsed as `"lorem_ipsum_dolor_sit_amet"`. This last string, if split with `"_"`, could be separated into `"lorem"` and `"ipsum_dolor_sit_amet"`, or `"lorem_ipsum_dolor"` and `"sit_amet"`. Hence the idea of the mapping to get the right separation.

Parameters

- **config_cls** (*type*) – the configuration class which will be used as a model to generate the options.
- **option_fields_mapping** (*dict, optional*) – a mapping of the option names, with POSIX convention (with "-" character"), to the list of fields: <option_name_with_dash>: <hierarchical_list_of_fields> This argument can be used to set the mapping directly, instead of creating it from a Configuration class.

add_arguments(*parser*)

Using the configuration class given, create automatically and recursively command-line options to set the different attributes of the configuration. Nested options keep the upper layers as prefixes, which are separated by a "-" character.

Generate the mapping between the option name and the hierarchy of the attributes of the Configuration.

Parameters **parser** (*argparse.ArgumentParser*) – the parser to which the new command-line options will be added.

merge(*config*, *args*)

Merge the configuration taken from file and the one from the command line arguments. The arguments have priority and replace the values read from configuration.

Parameters

- **config** (*dict*) – the configuration to replace the values from.
- **args** (*dict*) – the values read by the command line parser.

Returns

the result of the merge of the CLI arguments into the configuration, as serializable object.

Return type *krake.data.serializable.Serializable*

krake.**load_yaml_config**(*filepath*)

Load Krake base configuration settings from YAML file

Parameters **filepath** (*os.PathLike, optional*) – Path to YAML configuration file

Raises FileNotFoundError – If no configuration file can be found

Returns Krake YAML file configuration

Return type dict

krake.**search_config**(*filename*)

Search configuration file in known directories.

The filename is searched in the following directories in given order:

1. Current working directory
2. `/etc/krake`

Returns Path to configuration file

Return type os.PathLike

Raises `FileNotFoundError` – If the configuration cannot be found in any of the search locations.

`krake.`**`setup_logging`**(*config_log*)

Setups Krake logging based on logging configuration and global config level for each logger without log-level configuration

Parameters **`config_log`** (*dict*) – dictschema logging configuration (see `logging.config.dictConfig()`)

4.15.3 API Server

This module provides the HTTP RESTful API of the Krake application. It is implemented as an `aiohttp` application.

This module defines the bootstrap function for creating the aiohttp server instance serving Krake's HTTP API.

Krake serves multiple APIs for different technologies, e.g. the core functionality like roles and role bindings are served by the `krake.api.core` API where as the Kubernetes API is provided by `krake.api.kubernetes`.

Example

The API server can be run as follows:

```
from aiohttp import web
from krake.api.app import create_app

config = ...
app = create_app(config)
web.run_app(app)
```

`krake.api.app.`**`cors_setup`**(*app*)

Set the default CORS (Cross-Origin Resource Sharing) rules for all routes of the given web application.

Parameters **`app`** (*web.Application*) – Web application

`krake.api.app.`**`create_app`**(*config*)

Create aiohttp application instance providing the Krake HTTP API

Parameters **`config`** (*krake.data.config.ApiConfiguration*) – Application configuration object

Returns Krake HTTP API

Return type aiohttp.web.Application

`krake.api.app.`**`db_session`**(*app*, *host*, *port*)

Async generator creating a database *krake.api.database.Session* that can be used by other components (middleware, route handlers) or by the requests handlers. The database session is available under the `db` key of the application.

This function should be used as cleanup context (see `aiohttp.web.Application.cleanup_ctx`).

Parameters **`app`** (*aiohttp.web.Application*) – Web application

`krake.api.app.`**`http_session`**(*app*, *ssl_context=None*)

Async generator creating an `aiohttp.ClientSession` HTTP(S) session that can be used by other components (middleware, route handlers). The HTTP(S) client session is available under the `http` key of the application.

This function should be used as cleanup context (see aiohttp.web.Application.cleanup_ctx).

Parameters **app** (*aiohttp.web.Application*) – Web application

krake.api.app.**load_authentication**(*config*)
Create the authentication middleware *middlewares.authentication()*.

The authenticators are loaded from the "authentication" configuration key. If the server is configured with TLS, client certificates are also added as authentication (*auth.client_certificate_authentication()*) strategy.

Parameters **config** (*krake.data.config.ApiConfiguration*) – Application configuration object

Returns aiohttp middleware handling request authentication

krake.api.app.**load_authorizer**(*config*)
Load authorization function from configuration.

Parameters **config** (*krake.data.config.ApiConfiguration*) – Application configuration object

Raises ValueError – If an unknown authorization strategy is configured

Returns Coroutine function for authorizing resource requests

Authentication and Authorization

Authentication and Authorization module for Krake.

Access to the Krake API is controlled by two distinct mechanisms performed after each other:

Authentication verifies the identity of a user (Who is requesting?)

Authorization decides if the user has permission to access a resource

Authentication

Authentication is performed for every request. The *krake.api.middlewares.authentication()* middleware factory is used for this purpose. The concrete authentication implementation will be derived from the configuration.

```
# Anonymous authentication
authentication:
  kind: static
  name: system

# Keystone authentication
authentication:
  kind: keystone
  endpoint: http://localhost:5000/v3
```

An authenticator is a simple asynchronous function:

Currently, there are two authentication implementations available:

- Static authentication (*static_authentication()*)
- Keystone authentication (*keystone_authentication()*)

Authorization

Authorization is established with the help of the *protected()* decorator function. The decorator annotates a given aiohttp request handler with the required authorization information (see *AuthorizationRequest*).

An authorizer is a simple asynchronous function:

The concrete authentication implementation will be derived from the configuration and is stored under the `authorizer` key of the application.

```
# Authorization mode
#
#   - RBAC (Role-based access control)
#   - always-allow (Allow all requests. No authorization is performed.)
#   - always-deny (Deny all requests. Only for testing purposes.)
#
authorization: always-allow
```

Currently, there are three authorization implementations available:

- Always allow (*always_allow()*)
- Always deny (*always_deny()*)
- Role-based access control / RBAC (*rbac()*)

class krake.api.auth.**AuthorizationRequest**

Bases: tuple

Authorization request handled by authorizers.

api
Name of the API group

Type str

namespace
If the resource is namespaced, the requested namespace

Type str, optional

resource
Name of the resource

Type str

verb
Verb that should be performed on the resource.

Type *krake.data.core.Verb*

api
Alias for field number 0

namespace
Alias for field number 1

resource
Alias for field number 2

verb
Alias for field number 3

krake.api.auth.**always_allow**(*request*, *auth_request*)
Authorizer allowing every request.

Parameters

- **request** (*aiohttp.web.Request*) – Incoming HTTP request
- **auth_request** (AuthorizationRequest) – Authorization request associated with the incoming HTTP request.

krake.api.auth.**always_deny**(*request*, *auth_request*)
Authorizer denying every request.

Parameters

- **request** (*aiohttp.web.Request*) – Incoming HTTP request
- **auth_request** (AuthorizationRequest) – Authorization request associated with the incoming HTTP request.

Raises aiohttp.web.HTTPForbidden – Always raised

krake.api.auth.**client_certificate_authentication**()
Authenticator factory for authenticating requests with client certificates.

The client certificate is loaded from the peercert attribute of the underlying TCP transport. The common name of the client certificate is used as username

Returns Authenticator using client certificate information for authentication.

Return type callable

krake.api.auth.**keycloak_authentication**(*endpoint*, *realm*)
Authenticator factory for Keycloak authentication.

The token in the Authorization header of a request sent to Krake will be sent as access token to the OpenID user information endpoint. The returned user name from Keycloak is used as authenticated user name.

The authenticator requires an HTTP client session that is loaded from the http key of the application.

Parameters

- **endpoint** (*str*) – Keycloak HTTP endpoint.
- **realm** (*str*) – Keycloak realm to use at this endpoint.

Returns Authenticator for the given Keystone endpoint.

Return type callable

krake.api.auth.**keystone_authentication**(*endpoint*)
Authenticator factory for OpenStack Keystone authentication.

The token in the Authorization header of a request will be used as X-Auth-Token header for a request to the Keystone token endpoint. The returned user name from Keystone is used as authenticated user name.

The authenticator requires an HTTP client session that is loaded from the http key of the application.

Parameters **endpoint** (*str*) – Keystone HTTP endpoint

Returns Authenticator for the given Keystone endpoint.

Return type callable

krake.api.auth.**protected**(*api*, *resource*, *verb*)
Decorator function for aiohttp request handlers performing authorization.

The returned decorator can be used to wrap a given aiohttp handler and call the current authorizer of the application (loaded from the `authorizer` key of the application). If the authorizer does not raise any exception the request is authorized and the wrapped request handler is called.

Example

```
from krake.api.auth import protected

@routes.get("/book/{name}")
@protected(api="v1", resource="book", verb="get", namespaced=False)
async def get_resource(request):
    assert "user" in request
```

Parameters

- **api** (*str*) – Name of the API group
- **resource** (*str*) – Name of the resource
- **verb** (*str, krake.data.core.Verb*) – Verb that should be performed

Returns Decorator that can be used to wrap a given aiohttp request handler.

Return type callable

`krake.api.auth.`**rbac**(*request*, *auth_request*)

Role-based access control authorizer.

The roles of a user are loaded from the database. It checks if any role allows the verb on the resource in the namespace. Roles are only permissive. There are no denial rules.

Parameters

- **request** (*aiohttp.web.Request*) – Incoming HTTP request
- **auth_request** (AuthorizationRequest) – Authorization request associated with the incoming HTTP request.

Returns The role allowing access.

Return type *krake.data.core.Role*

Raises `aiohttp.web.HTTPForbidden` – If no role allows access.

`krake.api.auth.`**static_authentication**(*name*)

Authenticator factory for authenticating every request with the given name.

Parameters **name** (*str*) – Static user name that should be used for every request.

Returns Authenticator returning the given name for every request.

Return type callable

Database Abstraction

Database abstraction for etcd. Key idea of the abstraction is to provide an declarative way of defining persistent data structures (aka. "models") together with a simple interface for loading and storing these data structures.

This goal is achieved by leveraging the JSON-serializable data classes based on *krake.data.serializable* and combining them with a simple database session.

Example

```
from krake.api.database import Session
from krake.data import Key
from krake.data.serializable import Serializable


class Book(Serializable):
    isbn: int
    title: str

    __etcd_key__ = Key("/books/{isbn}")


async with Session(host="localhost") as session:
    book = await session.get(Book, isbn=9783453146976)
```

exception krake.api.database.**DatabaseError**
Bases: Exception

class krake.api.database.**EtcdClient**(*host='127.0.0.1'*, *port=2379*, *protocol='http'*, *cert=()*, *verify=None*, *timeout=None*, *headers=None*, *user_agent=None*, *pool_size=30*, *username=None*, *password=None*, *token=None*, *server_version='3.3.0'*, *cluster_version='3.3.0'*)
Bases: etcd3.aio_client.AioClient

Async etcd v3 client based on etcd3.aio_client.AioClient with some minor patches.

class krake.api.database.**Event**
Bases: tuple

Events that are yielded by *Session.watch()*

event
Type of event that occurred (PUT or DELETE)

Type *EventType*

value
Deserialized object. None if the event is of kind DELETE.

Type object, None

rev
Revision of the object

Type *Revision*

event
Alias for field number 0

rev
Alias for field number 2

value
Alias for field number 1

class krake.api.database.**EventType**
Bases: enum.Enum

Different types of events that can occur during *Session.watch()*.

class krake.api.database.**Revision**
Bases: tuple

Etcd revision of a loaded key-value pair.

Etcd stores all keys in a flat binary key space. The key space has a lexically sorted index on byte string keys. The key space maintains multiple revisions of the same key. Each atomic mutative operation (e.g., a transaction operation may contain multiple operations) creates a new revision on the key space.

Every *Session.get()* request returns also the revision besides the model.

key
Key in the etcd database

Type str

created
is the revision of last creation on this key.

Type int

modified
is the revision of last modification on this key.

Type int

version
is the version of the key. A deletion resets the version to zero and any modification of the key increases its version.

Type int

created
Alias for field number 1

key
Alias for field number 0

modified
Alias for field number 2

version
Alias for field number 3

class krake.api.database.**Session**(*host*, *port=2379*, *loop=None*)
Bases: object

Database session for managing *krake.data.serializable.Serializable* objects in an etcd database.

The serializable objects need have one additional attribute:

__etcd_key__ A *krake.data.Key* template for the associated etcd key of a managed object.

Objects managed by a session have an attached etcd *Revision* when loaded from the database. This revision can be read by *revision()*. If an object has no revision attached, it is considered *fresh* or *new*. It is expected that the associated key of a *new* object does not already exist in the database.

The session is an asynchronous context manager. It takes of care of opening and closing an HTTP session to the gRPC JSON gateway of the etcd server.

The etcd v3 protocol is documented by its protobuf definitions.

Example

```
async with Session(host="localhost") as session:
    pass
```

Parameters

- **host** (*str*) – Hostname of the etcd server
- **port** (*int, optional*) – Client port of the etcd server
- **loop** (*async.AbstractEventLoop, optional*) – asyncio event loop that should be used

all (*cls*, ***kwargs*)
Fetch all instances of a given type

The instances can be filtered by partial identities. Every identity can be specified as keyword argument and only instances with this identity attribute are returned. The only requirement for a filtered identity attribute is that all preceding identity attributes must also be given.

Example

```
class Book(Serializable):
    isbn: int
    title: str
    author: str

    __metadata__ = {
        "key": Key("/books/{author}/{isbn}")
    }

await db.all(Book)

# Get all books by Adam Douglas
await db.all(Book, author="Adam Douglas")

# This will raise a TypeError because the preceding "name"
# attribute is not given.
await db.all(Book, isbn=42)
```

Parameters

- **cls** (*type*) – Serializable class that should be loaded
- ****kwargs** – Parameters for the etcd key

Yields *(object, Revision)* – Tuple of deserialized model and revision

Raises TypeError – If an identity attribute is given without all preceding identity attributes.

client
Lazy loading of the etcd client. It is only created when the first request is performed.

Returns the client to connect to the database.

Return type *EtcdClient*

delete(*instance*)
Delete a given instance from etcd.

A transaction is used ensuring the etcd key was not modified in-between. If the transaction is successful, the revision of the instance will be updated to the revision returned by the transaction response.

Parameters **instance** (*object*) – Serializable object that should be deleted

Raises

- ValueError – If the passed object has no revision attached.
- *TransactionError* – If the key was modified in between

get(*cls*, ***kwargs*)
Fetch an serializable object from the etcd server specified by its identity attribute.

cls
Serializable class that should be loaded
Type type

****kwargs**
Parameters for the etcd key

Returns Deserialized model with attached revision. If the key was not found in etcd, None is returned.

Return type object, None

load_instance(*cls*, *kv*)
Load an instance and its revision by an etcd key-value pair

Parameters

- **cls** (*type*) – Serializable type
- **kv** – etcd key-value pair

Returns Deserialized model with attached revision

Return type object

put(*instance*)
Store new revision of a serializable object on etcd server.

If the instances does not have an attached *Revision* (see *revision()*), it is assumed that a key-value pair should be *created*. Otherwise, it is assumed that the key-value pair is updated.

A transaction ensures that

a) the etcd key was not modified in-between if the key is updated

b) the key does not already exists if a key is added

If the transaction is successful, the revision of the instance will updated to the revision returned by the transaction response.

Parameters

- **instance** (krake.data.serializable.Serializable) – Serializable object that
- **be stored.** (*should*) –

Raise:

TransactionError: If the key was modified in between or already exists

watch(*cls*, ***kwargs*)

Watch the namespace of a given serializable type and yield every change in this namespace.

Internally, it uses the etcd watch API. The `created` future can be used to signal successful creation of an etcd watcher.

Parameters

- **cls** (*type*) – Serializable type of which the namespace should be watched
- ****kwargs** – Parameters for the etcd key

Yields *Event* – Every change in the namespace will generate an event

exception krake.api.database.**TransactionError**

Bases: *krake.api.database.DatabaseError*

class krake.api.database.**Watcher**(*session*, *model*, ***kwargs*)

Bases: object

Async context manager for database watching requests.

This context manager is used internally by *Session.watch()*. It returns a async generator on entering. It is ensured that the watch is created on entering. This means inside the context, it can be assumed that the watch exists.

Parameters

- **session** (Session) – Database session doing the watch request
- **model** (*type*) – Class that is loaded from database
- ****kwargs** (*dict*) – Keyword arguments that are used to generate the etcd key prefix (`Key.prefix()`)

watch()

Async generator for watching database prefix.

Yields *Event* –

Database event holding the loaded model (see model argument) and database revision.

krake.api.database.**revision**(*instance*)

Returns the etcd *Revision* of an object used with a *Session*. If the object is currently *unattached* – which means it was not retrieved from the database with *Session.get()* – this function returns None.

Parameters **instance** (*object*) – Object used with *Session*.

Returns The current etcd revision of the instance.

Return type *Revision*, None

Helpers

Simple helper functions that are used by the HTTP endpoints.

class krake.api.helpers.**Heartbeat**(*response*, *interval=None*)

Bases: object

Asynchronous context manager for heartbeating long running HTTP responses.

Writes newlines to the response body in a given heartbeat interval. If `interval` is set to 0, no heartbeat will be sent.

Parameters

- **response** (*aiohttp.web.StreamResponse*) – Prepared HTTP response with chunked encoding
- **interval** (*int, float, optional*) – Heartbeat interval in seconds. Default: 10 seconds.

Raises ValueError – If the response is not prepared or not chunk encoded

Example

```
import asyncio
from aiohttp import web

from krake.helpers import Heartbeat


async def handler(request):
    # Prepare streaming response
    resp = web.StreamResponse()
    resp.enable_chunked_encoding()
    await resp.prepare(request)

    async with Heartbeat(resp):
        while True:
            await resp.write(b"spam\n")
            await asyncio.sleep(120)
```

heartbeat()
Indefinitely write a new line to the response body and sleep for `interval`.

class krake.api.helpers.**HttpProblem**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Store the reasons for failures of the HTTP layers for the API.

The reason is stored as an RFC 7807 Problem. It is a way to define a uniform, machine-readable details of errors in a HTTP response. See https://tools.ietf.org/html/rfc7807 for details.

type
A URI reference that identifies the problem type. It should point the Krake API users to the concrete part of the Krake documentation where the problem type is explained in detail. Defaults to about:blank.

Type str

title
A short, human-readable summary of the problem type

Type *HttpProblemTitle*

status
The HTTP status code

Type int

detail
A human-readable explanation of the problem

Type str

instance
A URI reference that identifies the specific occurrence of the problem

Type str

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

classmethod remove_none_values(*data*, ***kwargs*)
Remove attributes if value equals None

__post_init__()
HACK: marshmallow.Schema allows registering hooks like post_dump. This is not allowed in krake Serializable, therefore within marshmallow.Schema allows registering hooks like post_dump. This is not allowed in krake Serializable, therefore the __post_init__ method is registered directly within the hook.

remove_none_values(*data*, ***kwargs*)
Remove attributes if value equals None

exception krake.api.helpers.**HttpProblemError**(*exc: aiohttp.web_exceptions.HTTPException*, *problem: krake.api.helpers.HttpProblem = HttpProblem(type='about:blank', title=None, status=None, detail=None, instance=None)*, ***kwargs*)
Bases: Exception

Custom exception raised if failures on the HTTP layers occur

class krake.api.helpers.**HttpProblemTitle**
Bases: enum.Enum

Store the title of an RFC 7807 problem.

The RFC 7807 Problem title is a short, human-readable summary of the problem type. The name defines the title itself. The value is used as part of the URI reference that identifies the problem type, see *middlewares.problem_response()* for details.

class krake.api.helpers.**ListQuery**
Bases: object

Simple mixin class for operation template classes.

Defines default operation.query attribute for *list* and *list all* operations.

class krake.api.helpers.**QueryFlag**(***metadata*)
Bases: marshmallow.fields.Field

Field used for boolean query parameters.

If the query parameter exists the field is deserialized to True regardless of the value. The field is marked as load_only.

deserialize(*value*, *attr=None*, *data=None*, ***kwargs*)
Deserialize value.

Parameters

- **value** – The value to deserialize.

- **attr** – The attribute/key in *data* to deserialize.
- **data** – The raw input data passed to *Schema.load*.
- **kwargs** – Field-specific keyword arguments.

Raises **ValidationError** – If an invalid value is passed or if a required value is missing.

krake.api.helpers.**blocking**()

Decorator function to enable function blocking. This allows only a return of the response if the requested action is completed (eg. deletion of a resource). The function logic is therefore executed after its decorated counterpart.

Returns JSON style response coming from the handler

Return type Response

krake.api.helpers.**load**(*argname*, *cls*)

Decorator function for loading database models from URL parameters.

The wrapper loads the `name` parameter from the requests `match_info` attribute. If the `match_info` contains a `namespace` parameter, it is used as etcd key parameter as well.

Example

```
from aiohttp import web

from krake.data import serialize
from krake.data.core import Role

@load("role", Role)
def get_role(request, role):
    return json_response(serialize(role))
```

Parameters

- **argname** (*str*) – Name of the keyword argument that will be passed to the wrapped function.
- **cls** (*type*) – Database model class that should be loaded

Returns Decorator for aiohttp request handlers

Return type callable

krake.api.helpers.**make_create_request_schema**(*cls*)

Create a `marshmallow.Schema` excluding subresources and read-only.

Parameters **cls** (*type*) – Data class with `Schema` attribute

Returns Schema instance with excluded subresources

Return type marshmallow.Schema

krake.api.helpers.**session**(*request*)

Load the database session for a given aiohttp request

Internally, it just returns the value that was given as cleanup context by func:*krake.api.app.db_session*.

Parameters **request** (*aiohttp.web.Request*) – HTTP request

Returns Database session for the given request

Return type krake.database.Session

krake.api.helpers.**use_schema**(*argname*, *schema*)
Decorator function for loading a marshmallow.Schema from the request body.

If the request body is not valid JSON aiohttp.web.HTTPUnsupportedMediaType will be raised in the wrapper.

Parameters

- **argname** (*str*) – Name of the keyword argument that will be passed to the wrapped function.
- **schema** (*marshmallow.Schema*) – Schema that should used to deserialize the request body

Returns Decorator for aiohttp request handlers

Return type callable

Middlewares

This modules defines aiohttp middlewares for the Krake HTTP API

krake.api.middlewares.**authentication**(*authenticators*, *allow_anonymous*)
Middleware factory authenticating every request.

The concrete implementation is delegated to the passed asynchronous authenticator function (see *krake.api.auth* for details). This function returns the username for an incoming request. If the request is unauthenticated – meaning the authenticator returns None – system:anonymous is used as username.

The username is registered under the user key of the incoming request.

Anonymous requests can be allowed. If no authenticator authenticates the incoming request, "system:anonymous" is assigned as user for the request. This behavior can be disabled. In that case "401 Unauthorized" is raised if an request is not authenticated by any authenticator.

Parameters

- **authenticators** (*List[callable]*) – List if asynchronous function returning the username for a given request.
- **allow_anonymous** (*bool*) – If True, anonymous (unauthenticated) requests are allowed.

Returns aiohttp middleware loading a username for every incoming HTTP request.

krake.api.middlewares.**error_log**()
Middleware factory for logging exceptions in request handlers

Returns aiohttp middleware catching every exception logging it to the passed logger and reraising the exception.

krake.api.middlewares.**problem_response**(*problem_base_url=None*)
Middleware factory for HTTP exceptions in request handlers

Parameters **problem_base_url** (*str, optional*) – Base URL of the Krake documentation where HTTP problems are explained in detail.

Returns aiohttp middleware catching HttpProblemError or HTTPException based exception transforming the excpetion text to the *helpers.HttpProblem* (RFC 7807 Problem representation of failure) and reraising the exception.

krake.api.middlewares.**retry_transaction**(*retry=1*)
Middleware factory for transaction error handling.

If a *database.TransactionError* occurs, the request handler is retried for the specified number of times. If the transaction error persists, a *409 Conflict* HTTP exception is raised.

Parameters **retry** (*int, optional*) – Number of retries if a transaction error occurs.

Returns aiohttp middleware handling transaction errors.

Return type coroutine

4.15.4 Client

This module provides a simple Python client to the Krake HTTP API. It leverages the same data models as the API server from *krake.data*.

class krake.client.**ApiClient**(*client*)
Bases: object

Base class for all clients of a specific Krake API.

client
the lower-level client to use to create the actual connections.

Type *krake.client.Client*

plurals
contains the name of the resources handled by the current API and their corresponding names in plural: "<name_in_singular>": "<name_in_plural>"

Type dict[str, str]

Parameters **client** (krake.client.Client) – client to use for the HTTP communications.

class krake.client.**Client**(*url*, *loop=None*, *ssl_context=None*)
Bases: object

Simple async Python client for the Krake HTTP API.

The specific APIs are implemented in separate classes. Each API object requires an *Client* instance to interface the HTTP REST API.

The client implements the asynchronous context manager protocol used to handle opening and closing the internal HTTP session.

Example

```
from krake.client import Client
from krake.client.core import CoreApi

async with Client("http://localhost:8080") as client:
    core_api = CoreApi(client)
    role = await core_api.read_role(name="reader")
```

close()
Close the internal HTTP session and remove all resource attributes.

open()
Open the internal HTTP session and initializes all resource attributes.

class krake.client.**Watcher**(*session*, *url*, *model*)
Bases: object

Async context manager used by `watch_*()` methods of `ClientApi`.

The context manager returns the async generator of resources. On entering it is ensured that the watch is created. This means inside the context a watch is already established.

Parameters

- **session** (*aiohttp.ClientSession*) – HTTP session that is used to access the REST API.
- **url** (*str*) – URL for the watch request
- **model** (*type*) – Type that will be used to deserialize `krake.data.core.WatchEvent.object`

watch()
Async generator yielding watch events

Yields *krake.data.core.WatchEvent* –

Watch events where `object` is already deserialized correctly according to the API definition (see `model` argument)

Client APIs

class krake.client.core.**CoreApi**(*client*)
Bases: *krake.client.ApiClient*

Core API client

Example

```
from krake.client import Client

with Client(url="http://localhost:8080") as client:
    core_api = CoreApi(client)
```

Parameters **client** (krake.client.Client) – API client for accessing the Krake HTTP API

create_global_metric(*body*)
Create the specified GlobalMetric.

Parameters **body** (GlobalMetric) – Body of the HTTP request.

Returns Body of the HTTP response.

Return type *GlobalMetric*

create_global_metrics_provider(*body*)
Create the specified GlobalMetricsProvider.

Parameters **body** (GlobalMetricsProvider) – Body of the HTTP request.

Returns Body of the HTTP response.

Return type *GlobalMetricsProvider*

create_metric(*body*, *namespace*)
Create the specified Metric.

Parameters

- **body** (Metric) – Body of the HTTP request.
- **namespace** (*str*) – namespace of the Metric

Returns Body of the HTTP response.

Return type *Metric*

create_metrics_provider(*body*, *namespace*)
Create the specified MetricsProvider.

Parameters

- **body** (MetricsProvider) – Body of the HTTP request.
- **namespace** (*str*) – Namespace of the MetricsProvider.

Returns Body of the HTTP response.

Return type *MetricsProvider*

create_role(*body*)
Create the specified Role.

Parameters **body** (Role) – Body of the HTTP request.

Returns Body of the HTTP response.

Return type *Role*

create_role_binding(*body*)
Create the specified RoleBinding.

Parameters **body** (RoleBinding) – Body of the HTTP request.

Returns Body of the HTTP response.

Return type *RoleBinding*

delete_global_metric(*name*)
Delete the specified GlobalMetric.

Parameters **name** (*str*) – name of the GlobalMetric.

Returns Body of the HTTP response.

Return type *GlobalMetric*

delete_global_metrics_provider(*name*)
Delete the specified GlobalMetricsProvider.

Parameters **name** (*str*) – name of the GlobalMetricsProvider.

Returns Body of the HTTP response.

Return type *GlobalMetricsProvider*

delete_metric(*name*, *namespace*)
Delete the specified Metric.

Parameters

- **name** (*str*) – name of the Metric.

- **namespace** (*str*) – namespace of the Metric

Returns Body of the HTTP response.

Return type *Metric*

delete_metrics_provider(*name*, *namespace*)
Delete the specified MetricsProvider.

Parameters

- **name** (*str*) – name of the MetricsProvider.
- **namespace** (*str*) – namespace of the MetricsProvider.

Returns Body of the HTTP response.

Return type *MetricsProvider*

delete_role(*name*)
Delete the specified Role.

Parameters **name** (*str*) – name of the Role.

Returns Body of the HTTP response.

Return type *Role*

delete_role_binding(*name*)
Delete the specified RoleBinding.

Parameters **name** (*str*) – name of the RoleBinding.

Returns Body of the HTTP response.

Return type *RoleBinding*

list_global_metrics()
List the GlobalMetrics in the namespace.

Returns Body of the HTTP response.

Return type *GlobalMetricList*

list_global_metrics_providers()
List the GlobalMetricsProviders in the namespace.

Returns Body of the HTTP response.

Return type *GlobalMetricsProviderList*

list_metrics(*namespace=None*)
List the Metrics in the namespace.

Parameters **namespace** (*str*) – namespace of the Metric

Returns Body of the HTTP response.

Return type *MetricList*

list_metrics_providers(*namespace=None*)
List the MetricsProviders in the namespace.

Parameters **namespace** (*str*) – namespace of the MetricsProvider.

Returns Body of the HTTP response.

Return type *MetricsProviderList*

list_role_bindings()
List the RoleBindings in the namespace.

Returns Body of the HTTP response.

Return type *RoleBindingList*

list_roles()
List the Roles in the namespace.

Returns Body of the HTTP response.

Return type *RoleList*

read_global_metric(*name*)
Read the specified GlobalMetric.

Parameters **name** (*str*) – name of the GlobalMetric.

Returns Body of the HTTP response.

Return type *GlobalMetric*

read_global_metrics_provider(*name*)
Reads the specified GlobalMetricsProvider.

Parameters **name** (*str*) – name of the GlobalMetricsProvider.

Returns Body of the HTTP response.

Return type *GlobalMetricsProvider*

read_metric(*name*, *namespace*)
Read the specified Metric.

Parameters

- **name** (*str*) – name of the Metric.
- **namespace** (*str*) – namespace of the Metric

Returns Body of the HTTP response.

Return type *Metric*

read_metrics_provider(*name*, *namespace*)
Read the specified MetricsProvider.

Parameters

- **name** (*str*) – name of the MetricsProvider.
- **namespace** (*str*) – namespace of the MetricsProvider.

Returns Body of the HTTP response.

Return type *MetricsProvider*

read_role(*name*)
Read the specified Role.

Parameters **name** (*str*) – name of the Role.

Returns Body of the HTTP response.

Return type *Role*

read_role_binding(*name*)
Read the specified RoleBinding.

Parameters **name** (*str*) – name of the RoleBinding.

Returns Body of the HTTP response.

Return type *RoleBinding*

update_global_metric(*body*, *name*)
Update the specified GlobalMetric.

Parameters

- **body** (GlobalMetric) – Body of the HTTP request.
- **name** (*str*) – name of the GlobalMetric.

Returns Body of the HTTP response.

Return type *GlobalMetric*

update_global_metrics_provider(*body*, *name*)
Update the specified GlobalMetricsProvider.

Parameters

- **body** (GlobalMetricsProvider) – Body of the HTTP request.
- **name** (*str*) – name of the GlobalMetricsProvider.

Returns Body of the HTTP response.

Return type *GlobalMetricsProvider*

update_metric(*body*, *name*, *namespace*)
Update the specified GlobalMetric.

Parameters

- **body** (GlobalMetric) – Body of the HTTP request.
- **name** (*str*) – name of the Metric.
- **namespace** (*str*) – namespace of the Metric

Returns Body of the HTTP response.

Return type *GlobalMetric*

update_metrics_provider(*body*, *name*, *namespace*)
Update the specified MetricsProvider.

Parameters

- **body** (MetricsProvider) – Body of the HTTP request.
- **name** (*str*) – name of the MetricsProvider.
- **namespace** (*str*) – namespace of the MetricsProvider.

Returns Body of the HTTP response.

Return type *MetricsProvider*

update_role(*body*, *name*)
Update the specified Role.

Parameters

- **body** (Role) – Body of the HTTP request.
- **name** (*str*) – name of the Role.

Returns Body of the HTTP response.

Return type *Role*

update_role_binding(*body*, *name*)
Update the specified RoleBinding.

Parameters

- **body** (RoleBinding) – Body of the HTTP request.
- **name** (*str*) – name of the RoleBinding.

Returns Body of the HTTP response.

Return type *RoleBinding*

watch_global_metrics(*heartbeat=None*)
Generate a watcher for the GlobalMetrics in the namespace.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *GlobalMetricList*

watch_global_metrics_providers(*heartbeat=None*)
Generate a watcher for the GlobalMetricsProviders in the namespace.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *GlobalMetricsProviderList*

watch_metrics(*namespace=None*, *heartbeat=None*)
Generate a watcher for the Metrics in the namespace.

Parameters

- **namespace** (*str*) – namespace of the Metric
- **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *MetricList*

watch_metrics_providers(*namespace=None*, *heartbeat=None*)
Generate a watcher for the MetricsProviders in the namespace.

Parameters

- **namespace** (*str*) – namespace of the MetricsProvider.
- **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *MetricsProviderList*

watch_role_bindings(*heartbeat=None*)
Generate a watcher for the RoleBindings in the namespace.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *RoleBindingList*

watch_roles(*heartbeat=None*)
Generate a watcher for the Roles in the namespace.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *RoleList*

class krake.client.infrastructure.**InfrastructureApi**(*client*)
Bases: *krake.client.ApiClient*

Infrastructure API client

Example

```
from krake.client import Client

with Client(url="http://localhost:8080") as client:
    infrastructure_api = InfrastructureApi(client)
```

Parameters **client** (krake.client.Client) – API client for accessing the Krake HTTP API

create_cloud(*body*, *namespace*)
Create the specified Cloud.

Parameters

- **body** (Cloud) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Cloud will be updated.

Returns Body of the HTTP response.

Return type *Cloud*

create_global_cloud(*body*)
Create the specified GlobalCloud.

Parameters **body** (GlobalCloud) – Body of the HTTP request.

Returns Body of the HTTP response.

Return type *GlobalCloud*

create_global_infrastructure_provider(*body*)
Create the specified GlobalInfrastructureProvider.

Parameters **body** (GlobalInfrastructureProvider) – Body of the HTTP request.

Returns Body of the HTTP response.

Return type *GlobalInfrastructureProvider*

create_infrastructure_provider(*body*, *namespace*)
Create the specified InfrastructureProvider.

Parameters

- **body** (InfrastructureProvider) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the InfrastructureProvider will be updated.

Returns Body of the HTTP response.

Return type *InfrastructureProvider*

delete_cloud(*namespace*, *name*)
Delete the specified Cloud.

Parameters

- **namespace** (*str*) – namespace in which the Cloud will be updated.
- **name** (*str*) – name of the Cloud.

Returns Body of the HTTP response.

Return type *Cloud*

delete_global_cloud(*name*)
Delete the specified GlobalCloud.

Parameters **name** (*str*) – name of the GlobalCloud.

Returns Body of the HTTP response.

Return type *GlobalCloud*

delete_global_infrastructure_provider(*name*)
Delete the specified GlobalInfrastructureProvider.

Parameters **name** (*str*) – name of the GlobalInfrastructureProvider.

Returns Body of the HTTP response.

Return type *GlobalInfrastructureProvider*

delete_infrastructure_provider(*namespace*, *name*)
Delete the specified InfrastructureProvider.

Parameters

- **namespace** (*str*) – namespace in which the InfrastructureProvider will be updated.
- **name** (*str*) – name of the InfrastructureProvider.

Returns Body of the HTTP response.

Return type *InfrastructureProvider*

list_all_clouds()
List all Clouds.

Returns Body of the HTTP response.

Return type *CloudList*

list_all_infrastructure_providers()
List all InfrastructureProviders.

Returns Body of the HTTP response.

Return type *InfrastructureProviderList*

list_clouds(*namespace*)
List the Clouds in the namespace.

Parameters **namespace** (*str*) – namespace in which the Cloud will be updated.

Returns Body of the HTTP response.

Return type *CloudList*

list_global_clouds()
List the GlobalClouds in the namespace.

Returns Body of the HTTP response.

Return type *GlobalCloudList*

list_global_infrastructure_providers()
List the GlobalInfrastructureProviders in the namespace.

Returns Body of the HTTP response.

Return type *GlobalInfrastructureProviderList*

list_infrastructure_providers(*namespace*)
List the InfrastructureProviders in the namespace.

Parameters **namespace** (*str*) – namespace in which the InfrastructureProvider will be updated.

Returns Body of the HTTP response.

Return type *InfrastructureProviderList*

read_cloud(*namespace*, *name*)
Read the specified Cloud.

Parameters

- **namespace** (*str*) – namespace in which the Cloud will be updated.
- **name** (*str*) – name of the Cloud.

Returns Body of the HTTP response.

Return type *Cloud*

read_global_cloud(*name*)
Read the specified GlobalCloud.

Parameters **name** (*str*) – name of the GlobalCloud.

Returns Body of the HTTP response.

Return type *GlobalCloud*

read_global_infrastructure_provider(*name*)
Read the specified GlobalInfrastructureProvider.

Parameters **name** (*str*) – name of the GlobalInfrastructureProvider.

Returns Body of the HTTP response.

Return type *GlobalInfrastructureProvider*

read_infrastructure_provider(*namespace*, *name*)
Read the specified InfrastructureProvider.

Parameters

- **namespace** (*str*) – namespace in which the InfrastructureProvider will be updated.
- **name** (*str*) – name of the InfrastructureProvider.

Returns Body of the HTTP response.

Return type *InfrastructureProvider*

update_cloud(*body*, *namespace*, *name*)
Update the specified Cloud.

Parameters

- **body** (Cloud) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Cloud will be updated.
- **name** (*str*) – name of the Cloud.

Returns Body of the HTTP response.

Return type *Cloud*

update_cloud_status(*body*, *namespace*, *name*)
Update the specified Cloud.

Parameters

- **body** (Cloud) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Cloud will be updated.
- **name** (*str*) – name of the Cloud.

Returns Body of the HTTP response.

Return type *Cloud*

update_global_cloud(*body*, *name*)
Update the specified GlobalCloud.

Parameters

- **body** (GlobalCloud) – Body of the HTTP request.
- **name** (*str*) – name of the GlobalCloud.

Returns Body of the HTTP response.

Return type *GlobalCloud*

update_global_cloud_status(*body*, *name*)
Update the specified GlobalCloud.

Parameters

- **body** (GlobalCloud) – Body of the HTTP request.
- **name** (*str*) – name of the GlobalCloud.

Returns Body of the HTTP response.

Return type *GlobalCloud*

update_global_infrastructure_provider(*body*, *name*)
Update the specified GlobalInfrastructureProvider.

Parameters

- **body** (GlobalInfrastructureProvider) – Body of the HTTP request.
- **name** (*str*) – name of the GlobalInfrastructureProvider.

Returns Body of the HTTP response.

Return type *GlobalInfrastructureProvider*

update_infrastructure_provider(*body*, *namespace*, *name*)
Update the specified InfrastructureProvider.

Parameters

- **body** (InfrastructureProvider) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the InfrastructureProvider will be updated.
- **name** (*str*) – name of the InfrastructureProvider.

Returns Body of the HTTP response.

Return type *InfrastructureProvider*

watch_all_clouds(*heartbeat=None*)
Generate a watcher for all Clouds.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *CloudList*

watch_all_infrastructure_providers(*heartbeat=None*)
Generate a watcher for all InfrastructureProviders.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *InfrastructureProviderList*

watch_clouds(*namespace*, *heartbeat=None*)
Generate a watcher for the Clouds in the namespace.

Parameters

- **namespace** (*str*) – namespace in which the Cloud will be updated.
- **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *CloudList*

watch_global_clouds(*heartbeat=None*)
Generate a watcher for the GlobalClouds in the namespace.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *GlobalCloudList*

watch_global_infrastructure_providers(*heartbeat=None*)
Generate a watcher for the GlobalInfrastructureProviders in the namespace.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *GlobalInfrastructureProviderList*

watch_infrastructure_providers(*namespace*, *heartbeat=None*)
Generate a watcher for the InfrastructureProviders in the namespace.

Parameters

- **namespace** (*str*) – namespace in which the InfrastructureProvider will be updated.
- **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type *InfrastructureProviderList*

class krake.client.kubernetes.**KubernetesApi**(*client*)
Bases: *krake.client.ApiClient*

Kubernetes API client

Example

```
from krake.client import Client

with Client(url="http://localhost:8080") as client:
    kubernetes_api = KubernetesApi(client)
```

Parameters **client** (krake.client.Client) – API client for accessing the Krake HTTP API

create_application(*body*, *namespace*)
Creates the specified Application.

Parameters

- **body** (Application) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Application will be updated.

Returns Body of the HTTP response.

Return type *Application*

create_cluster(*body*, *namespace*)
Creates the specified Cluster.

Parameters

- **body** (Cluster) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Cluster will be updated.

Returns Body of the HTTP response.

Return type *Cluster*

delete_application(*namespace*, *name*)
Deletes the specified Application.

Parameters

- **namespace** (*str*) – namespace in which the Application will be updated.
- **name** (*str*) – name of the Application.

Returns Body of the HTTP response.

Return type *Application*

delete_cluster(*namespace*, *name*)
Deletes the specified Cluster.

Parameters

- **namespace** (*str*) – namespace in which the Cluster will be updated.
- **name** (*str*) – name of the Cluster.

Returns Body of the HTTP response.

Return type *Cluster*

list_all_applications()
Lists all Applications.

Returns Body of the HTTP response.

Return type *ApplicationList*

list_all_clusters()
Lists all Clusters.

Returns Body of the HTTP response.

Return type *ClusterList*

list_applications(*namespace*)
Lists the Applications in the namespace.

Parameters **namespace** (*str*) – namespace in which the Application will be updated.

Returns Body of the HTTP response.

Return type *ApplicationList*

list_clusters(*namespace*)
Lists the Clusters in the namespace.

Parameters **namespace** (*str*) – namespace in which the Cluster will be updated.

Returns Body of the HTTP response.

Return type *ClusterList*

read_application(*namespace*, *name*)
Reads the specified Application.

Parameters

- **namespace** (*str*) – namespace in which the Application will be updated.
- **name** (*str*) – name of the Application.

Returns Body of the HTTP response.

Return type *Application*

read_cluster(*namespace*, *name*)
Reads the specified Cluster.

Parameters

- **namespace** (*str*) – namespace in which the Cluster will be updated.
- **name** (*str*) – name of the Cluster.

Returns Body of the HTTP response.

Return type *Cluster*

update_application(*body*, *namespace*, *name*)
Updates the specified Application.

Parameters

- **body** (Application) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Application will be updated.
- **name** (*str*) – name of the Application.

Returns Body of the HTTP response.

Return type *Application*

update_application_binding(*body*, *namespace*, *name*)
Updates the specified Application.

Parameters

- **body** (ClusterBinding) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Application will be updated.
- **name** (*str*) – name of the Application.

Returns Body of the HTTP response.

Return type *Application*

update_application_complete(*body*, *namespace*, *name*)
Updates the specified Application.

Parameters

- **body** (ApplicationComplete) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Application will be updated.
- **name** (*str*) – name of the Application.

Returns Body of the HTTP response.

Return type *Application*

update_application_shutdown(*body*, *namespace*, *name*)
Updates the specified Application.

Parameters

- **body** (ApplicationShutdown) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Application will be updated.
- **name** (*str*) – name of the Application.

Returns Body of the HTTP response.

Return type *Application*

update_application_status(*body*, *namespace*, *name*)
Updates the specified Application.

Parameters

- **body** (Application) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Application will be updated.
- **name** (*str*) – name of the Application.

Returns Body of the HTTP response.

Return type *Application*

update_cluster(*body*, *namespace*, *name*)
Updates the specified Cluster.

Parameters

- **body** (Cluster) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Cluster will be updated.
- **name** (*str*) – name of the Cluster.

Returns Body of the HTTP response.

Return type *Cluster*

update_cluster_binding(*body*, *namespace*, *name*)
Update the specified Cluster.

Parameters

- **body** (CloudBinding) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Cluster will be updated.
- **name** (*str*) – name of the Cluster.

Returns Body of the HTTP response.

Return type *Cluster*

update_cluster_status(*body*, *namespace*, *name*)
Updates the specified Cluster.

Parameters

- **body** (Cluster) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Cluster will be updated.
- **name** (*str*) – name of the Cluster.

Returns Body of the HTTP response.

Return type *Cluster*

watch_all_applications(*heartbeat=None*)
Generates a watcher for all Applications.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds.

Returns Body of the HTTP response.

Return type *ApplicationList*

watch_all_clusters(*heartbeat=None*)
Generates a watcher for all Clusters.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds.

Returns Body of the HTTP response.

Return type *ClusterList*

watch_applications(*namespace*, *heartbeat=None*)
Generates a watcher for the Applications in the namespace.

Parameters

- **namespace** (*str*) – namespace in which the Application will be updated.
- **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds.

Returns Body of the HTTP response.

Return type *ApplicationList*

watch_clusters(*namespace*, *heartbeat=None*)
Generates a watcher for the Clusters in the namespace.

Parameters

- **namespace** (*str*) – namespace in which the Cluster will be updated.
- **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds.

Returns Body of the HTTP response.

Return type *ClusterList*

class krake.client.openstack.**OpenStackApi**(*client*)
Bases: *krake.client.ApiClient*

Openstack API client

Example

```
from krake.client import Client

with Client(url="http://localhost:8080") as client:
    openstack_api = OpenStackApi(client)
```

Parameters **client** (krake.client.Client) – API client for accessing the Krake HTTP API

create_magnum_cluster(*body*, *namespace*)
Creates the specified MagnumCluster.

Parameters

- **body** (*MagnumCluster*) – Body of the HTTP request.

- **namespace** (*str*) – namespace in which the MagnumCluster will be updated.

Returns Body of the HTTP response.

Return type MagnumCluster

create_project(*body*, *namespace*)
Creates the specified Project.

Parameters

- **body** (*Project*) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Project will be updated.

Returns Body of the HTTP response.

Return type Project

delete_magnum_cluster(*namespace*, *name*)
Deletes the specified MagnumCluster.

Parameters

- **namespace** (*str*) – namespace in which the MagnumCluster will be updated.
- **name** (*str*) – name of the MagnumCluster.

Returns Body of the HTTP response.

Return type MagnumCluster

delete_project(*namespace*, *name*)
Deletes the specified Project.

Parameters

- **namespace** (*str*) – namespace in which the Project will be updated.
- **name** (*str*) – name of the Project.

Returns Body of the HTTP response.

Return type Project

list_all_magnum_clusters()
Lists all MagnumClusters.

Returns Body of the HTTP response.

Return type MagnumClusterList

list_all_projects()
Lists all Projects.

Returns Body of the HTTP response.

Return type ProjectList

list_magnum_clusters(*namespace*)
Lists the MagnumClusters in the namespace.

Parameters **namespace** (*str*) – namespace in which the MagnumCluster will be updated.

Returns Body of the HTTP response.

Return type MagnumClusterList

list_projects(*namespace*)
Lists the Projects in the namespace.

Parameters **namespace** (*str*) – namespace in which the Project will be updated.

Returns Body of the HTTP response.

Return type ProjectList

read_magnum_cluster(*namespace*, *name*)
Reads the specified MagnumCluster.

Parameters

- **namespace** (*str*) – namespace in which the MagnumCluster will be updated.
- **name** (*str*) – name of the MagnumCluster.

Returns Body of the HTTP response.

Return type MagnumCluster

read_project(*namespace*, *name*)
Reads the specified Project.

Parameters

- **namespace** (*str*) – namespace in which the Project will be updated.
- **name** (*str*) – name of the Project.

Returns Body of the HTTP response.

Return type Project

update_magnum_cluster(*body*, *namespace*, *name*)
Updates the specified MagnumCluster.

Parameters

- **body** (*MagnumCluster*) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the MagnumCluster will be updated.
- **name** (*str*) – name of the MagnumCluster.

Returns Body of the HTTP response.

Return type MagnumCluster

update_magnum_cluster_binding(*body*, *namespace*, *name*)
Updates the specified MagnumCluster.

Parameters

- **body** (*MagnumClusterBinding*) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the MagnumCluster will be updated.
- **name** (*str*) – name of the MagnumCluster.

Returns Body of the HTTP response.

Return type MagnumCluster

update_magnum_cluster_status(*body*, *namespace*, *name*)
Updates the specified MagnumCluster.

Parameters

- **body** (*MagnumCluster*) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the MagnumCluster will be updated.
- **name** (*str*) – name of the MagnumCluster.

Returns Body of the HTTP response.

Return type MagnumCluster

update_project(*body*, *namespace*, *name*)
Updates the specified Project.

Parameters

- **body** (*Project*) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Project will be updated.
- **name** (*str*) – name of the Project.

Returns Body of the HTTP response.

Return type Project

update_project_status(*body*, *namespace*, *name*)
Updates the specified Project.

Parameters

- **body** (*Project*) – Body of the HTTP request.
- **namespace** (*str*) – namespace in which the Project will be updated.
- **name** (*str*) – name of the Project.

Returns Body of the HTTP response.

Return type Project

watch_all_magnum_clusters(*heartbeat=None*)
Generates a watcher for all MagnumClusters.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type MagnumClusterList

watch_all_projects(*heartbeat=None*)
Generates a watcher for all Projects.

Parameters **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type ProjectList

watch_magnum_clusters(*namespace*, *heartbeat=None*)
Generates a watcher for the MagnumClusters in the namespace.

Parameters

- **namespace** (*str*) – namespace in which the MagnumCluster will be updated.
- **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type MagnumClusterList

watch_projects(*namespace*, *heartbeat=None*)
Generates a watcher for the Projects in the namespace.

Parameters

- **namespace** (*str*) – namespace in which the Project will be updated.
- **heartbeat** (*int*) – Number of seconds after which the server sends a heartbeat in form of an empty newline. Passing 0 disables the heartbeat. Default: 10 seconds

Returns Body of the HTTP response.

Return type ProjectList

4.15.5 Controllers

This module comprises Krake controllers responsible for watching API resources and transferring the state of related real-world resources to the desired state specified in the API. Controllers can be written in any language and with every technique. This module provides basic functionality and paradigms to implement a simple "control loop mechanism" in Python.

class krake.controller.**BurstWindow**(*name*, *burst_time*, *max_retry=0*, *loop=None*)
Bases: object

Context manager that can be used to check the time arbitrary code took to run. This arbitrary code should be something that needs to run indefinitely. If this code fails too quickly, it is not restarted.

The criteria are as follows: every `max_retry` times, if the average running time of the task is more than the `burst_time`, the task is considered savable and the context manager is exited. If not, an exception will be raised.

```
window = BurstWindow("my_task", 10, max_retry=3)

while True:  # use any kind of loop
    with window:
        # code to retry
        # ...
```

Parameters

- **name** (*str*) – the name of the background task (for debugging purposes).
- **burst_time** (*float*) – maximal accepted average time for a retried task.
- **max_retry** (*int, optional*) – number of times the task should be retried before testing the burst time. If 0, the task will be retried indefinitely, without looking for attr:*burst_time*.
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.

__exit__(**exc*)
After the given number of tries, raise an exception if the content of the context manager failed too fast.

Raises RuntimeError – if a background task keep on failing more regularly than what the burst time allows.

class krake.controller.**Controller**(*api_endpoint*, *loop=None*, *ssl_context=None*, *debounce=0*)
Bases: object

Base class for Krake controllers providing basic functionality for watching and enqueuing API resources.

The basic workflow is as follows: the controller holds several background tasks. The API resources are watched by a Reflector, which calls a handler on each received state of a resource. Any received new state is put into a *WorkQueue*. Multiple workers consume this queue. Workers are responsible for doing the actual state transitions. The work queue ensures that a resource is processed by one worker at a time (strict sequential). The status of the real world resources is monitored by an Observer (another background task).

However, this workflow is just a possibility. By modifying `__init__()` (or other functions), it is possible to add other queues, change the workers at will, add several Reflector or Observer, create additional background tasks...

Parameters

- **api_endpoint** (*str*) – URL to the API
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.
- **ssl_context** (*ssl.SSLContext, optional*) – if given, this context will be used to communicate with the API endpoint.
- **debounce** (*float, optional*) – value of the debounce for the *WorkQueue*.

cleanup()
Unregister all background tasks that are attributes.

create_endpoint(*api_endpoint*)
Ensure the scheme (HTTP/HTTPS) of the endpoint to connect to the API, depending on the existence of a given SSL context.

Parameters **api_endpoint** (*str*) – the given API endpoint.

Returns the final endpoint with the right scheme.

Return type str

prepare(*client*)
Start all API clients that the controller will be using. Create all necessary coroutines and register them as background tasks that will be started by the Controller.

Parameters **client** (krake.client.Client) – the base client to use for the API client to connect to the API.

register_task(*corofactory*, *name=None*)

Add a coroutine to the list of task that will be run in the background of the Controller.

Parameters

- **corofactory** (*coroutine*) – the coroutine that will be used as task. It must be running indefinitely and not catch asyncio.CancelledError.
- **name** (*str, optional*) – the name of the background task, for logging purposes.

retry(*coro*, *name=''*)
Start a background task. If the task fails not too regularly, restart it A *BurstWindow* is used to decide if the task should be restarted.

Parameters

- **coro** (*coroutine*) – the background task to try to restart.
- **name** (*str*) – the name of the background task (for debugging purposes).

Raises RuntimeError – if a background task keep on failing more regularly than what the burst time allows.

run()
Start at once all the registered background tasks with the retry logic.

simple_on_receive(*resource*, *condition=<class 'bool'>*)
Example of a resource receiving handler, that accepts a resource under conditions, and if they are met, add the resource to the queue. When listing values, you get a Resource, while when watching, you get an Event.

Parameters

- **resource** (krake.data.serializable.Serializable) – a resource received by listing.
- **condition** (*callable, optional*) – a condition to accept the given resource. The signature should be `(resource) -> bool`.

exception krake.controller.**ControllerError**(*message*)
Bases: Exception

Base class for exceptions during handling of a resource.

__str__()
Custom error message for exception

class krake.controller.**Executor**(*controller*, *loop=None*, *catch_signals=True*)
Bases: object

Component used to encapsulate the Controller. It takes care of starting the Controller, and handles all logic not directly dependent to the Controller, such as the handlers for the UNIX signals.

It implements the asynchronous context manager protocol. The controller itself can be awaited. The "await" call blocks until the Controller terminates.

```
executor = Executor(controller)
async with executor:
    await executor
```

Parameters

- **controller** (krake.controller.Controller) – the controller that the executor is tasked with starting.
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.
- **catch_signals** (*bool, optional*) – if True, the Executor will add handlers to catch killing signals in order to stop the Controller and the Executor gracefully.

__aenter__()
Create the signal handlers and start the Controller as background task.

__aexit__(**exc*)
Wait for the managed controller to be finished and cleanup.

stop()
Called as signal handler. Stop the Controller managed by the instance.

class krake.controller.**Observer**(*resource*, *on_res_update*, *time_step=1*)
Bases: object

Component used to watch the actual status of one instance of any resource.

Parameters

- **resource** – the instance of a resource that the Observer has to watch.
- **on_res_update** (*coroutine*) – a coroutine called when a resource's actual status differs from the status sent by the database. Its signature is: `(resource) -> updated_resource`. `updated_resource` is the instance of the resource that is up-to-date with the API. The Observer internal instance of the resource to observe will be updated. If the API cannot be contacted, `None` can be returned. In this case the internal instance of the Observer will not be updated.
- **time_step** (*int, optional*) – how frequently the Observer should watch the actual status of the resources.

observe_resource()
Update the watched resource if its status is different from the status observed. The status sent for the update is the observed one.

poll_resource()
Fetch the current status of the watched resource.

Returns

Return type *krake.data.core.Status*

run()
Start the observing process indefinitely, with the Observer time step.

class krake.controller.**Reflector**(*listing*, *watching*, *on_list=None*, *on_add=None*, *on_update=None*, *on_delete=None*, *resource_plural=None*, *loop=None*)
Bases: object

Component used to contact the API, fetch resources and handle disconnections.

Parameters

- **listing** (*coroutine*) – the coroutine used to get the list of resources currently stored by the API. Its signature is: `() -> <Resource>List`.
- **watching** (*coroutine*) – the coroutine used to watch updates on the resources, as sent by the API. Its signature is: `() -> watching object`. This watching object should be able to be used as context manager, and as generator.
- **on_list** (*coroutine*) – the coroutine called when listing all resources with the fetched resources as parameter. Its signature is: `(resource) -> None`.
- **on_add** (*coroutine, optional*) – the coroutine called during watch, when an ADDED event has been received. Its signature is: `(resource) -> None`.
- **on_update** (*coroutine, optional*) – the coroutine called during watch, when a MODIFIED event has been received. Its signature is: `(resource) -> None`.
- **on_delete** (*coroutine, optional*) – the coroutine called during watch, when a DELETED event has been received. Its signature is: `(resource) -> None`.
- **resource_plural** (*str, optional*) – name of the resource that the reflector is monitoring. For logging purpose. Default is `"resources"`

- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.

__call__(*min_interval=2*)
Start the Reflector. Encapsulate the connections with a retry logic, as disconnections are expected. If any other kind of error occurs, they are not swallowed.

Between two connection attempts, the connection will be retried later with a delay. If the connection fails to fast, the delay will be increased, to wait for the API to be ready. If the connection succeeded for a certain interval, the value of the delay is reset.

Parameters **min_interval** (*int, optional*) – if the connection was kept longer than this value, the delay is reset to the base value, as it is considered that a connection was possible.

list_and_watch()
Start the given list and watch coroutines.

list_resource()
Pass each resource returned by the current instance's listing function as parameter to the receiving function.

watch_resource(*watcher*)
Pass each resource returned by the current instance's watching object as parameter to the event receiving functions.

Parameters **watcher** – an object that returns a new event every time an update on a resource occurs

class krake.controller.**WorkQueue**(*maxsize=0*, *debounce=0*, *loop=None*)
Bases: object

Simple asynchronous work queue.

The key manages a set of key-value pairs. The queue guarantees strict sequential processing of keys: A key-value pair retrieved via *get()* is not returned via *get()* again until *done()* with the corresponding key is called, even if a new key-value pair with the corresponding key was put into the queue during the time of processing.

Parameters

- **maxsize** (*int, optional*) – Maximal number of items in the queue before *put()* blocks. Defaults to 0 which means the size is infinite
- **debounce** (*float*) – time in second for the debouncing of the values. A number higher than 0 means that the queue will wait the given time before giving a value. If a newer value is received, this time is reset.
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used

`dirty` holds the last known value of a key i.e. the next value which will be given by the *get()* method.

`timers` holds the current debounce coroutine for a key. Either this coroutine is canceled (if a new value for a key is given to the WorkQueue through the meth:*put*) or the value is added to the `dirty` dictionary.

`active` ensures that a key isn't added twice to the `queue`. Keys are added to this set when they are first added to the `dirty` dictionary, and are removed from the set when the Worker calls the *done()* method.

Todo:

- Implement rate limiting and delays

---

cancel(*key*)
Cancel the corresponding debounce coroutine for the given key. An attempt to cancel the coroutine for a key which was not inserted into the queue does not raise any error, and is simply ignored.

Parameters **key** – Key that identifies the value

close()
Cancel all pending debounce timers.

done(*key*)
Called by the Worker to notify that the work on the given key is done. This method first removes the key from the `active` set, and then adds this key to the set if a new value has arrived.

Parameters **key** – Key that used to identity the value

empty()
Check if the queue is empty

Returns bool: True if there are no dirty keys

full()
Check if the queue is full

Returns True if the queue is full

Return type bool

get()
Retrieve a key-value pair from the queue.

The queue will not return this key as long as *`done()`* is not called with this key.

Returns (key, value) tuple

put(*key*, *value*, *delay=None*)
Put a new key-value pair into the queue.

Parameters

- **key** – Key that used to identify the value
- **value** – New value that is associated with the key
- **delay** (*`float, optional`*) – Number of seconds the put should be delayed. If `None` is given, `debounce` will be used.

size()
Returns the number of keys marked as "dirty"

Returns Number of dirty keys in the queue

Return type int

krake.controller.**create_ssl_context**(*tls_config*)
From a certificate, create an SSL Context that can be used on the client side for communicating with a Server.

Parameters **tls_config** (*`krake.data.config.TlsClientConfiguration`*) – the "tls" configuration part of a controller.

Returns a default SSL Context tweaked with the given certificate elements

Return type ssl.SSLContext

krake.controller.**joint**(**aws*, *loop=None*)
Start several coroutines together. Ensure that if one stops, all others are cancelled as well.

FIXME: using asyncio.gather, if an error occurs in one of the "gathered" task, all the tasks are not necessarily stopped. @see https://stackoverflow.com/questions/59073556/how-to-cancel-all-remaining-tasks-in-gather-if-one-fails # noqa

Parameters

- **aws** (*Awaitable*) – a list of await-ables to start concurrently.
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.

krake.controller.**run**(*controller*)

Start the controller using an executor.

Parameters **controller** (krake.controller.Controller) – the controller to start

krake.controller.**sigmoid_delay**(*retries*, *maximum=60.0*, *steepness=0.75*, *midpoint=10.0*, *base=1.0*)

Compute a waiting time (delay) depending on the number of retries already performed. The computing function is a sigmoid.

Parameters

- **retries** (*int*) – the number of attempts that happened already.
- **maximum** (*float*) – the maximum delay that can be attained. Maximum of the sigmoid.
- **steepness** (*float*) – how fast the delay increases. Steepness of the sigmoid.
- **midpoint** (*float*) – number of retries to reach the delay between maximum and base. Midpoint of the sigmoid.
- **base** (*float*) – minimum value for the delay.

Returns the computed next delay.

Return type float

Controller Kubernetes Application

Module comprises Krake Kubernetes application controller logic.

class krake.controller.kubernetes.application.**KubernetesApplicationController**(*api_endpoint*, *worker_count=10*, *loop=None*, *ssl_context=None*, *debounce=0*, *hooks=None*, *time_step=2*)

Bases: *krake.controller.Controller*

Controller responsible for *krake.data.kubernetes.Application* resources. The controller manages Application resources in "SCHEDULED" and "DELETING" state.

kubernetes_api

Krake internal API to connect to the "kubernetes" API of Krake.

Type *KubernetesApi*

application_reflector

reflector for the Application resource of the

Type *Reflector*

"kubernetes" API of Krake.

worker_count
the amount of worker function that should be run as background tasks.

Type int

hooks
configuration to be used by the hooks supported by the controller.

Type krake.data.config.HooksConfiguration

observer_time_step
for the Observers: the number of seconds between two observations of the actual resource.

Type float

observers
mapping of all Application resource' UID to their respective Observer and task responsible for the Observer. The signature is: `<uid> --> <observer>, <reference_to_observer's_task>`.

Type dict[str, (*Observer*, Coroutine)]

Parameters

- **api_endpoint** (*str*) – URL to the API
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.
- **ssl_context** (*ssl.SSLContext, optional*) – if given, this context will be used to communicate with the API endpoint.
- **debounce** (*float, optional*) – value of the debounce for the `WorkQueue`.
- **worker_count** (*int, optional*) – the amount of worker function that should be run as background tasks.
- **time_step** (*float, optional*) – for the Observers: the number of seconds between two observations of the actual resource.

check_external_endpoint()
Ensure the scheme in the external endpoint (if provided) is matching the scheme used by the Krake API ("https" or "http" if TLS is enabled or disabled respectively).

If they are not, a warning is logged and the scheme is replaced in the endpoint.

cleanup()
Unregister all background tasks that are attributes.

handle_resource(*run_once=False*)
Infinite loop which fetches and hand over the resources to the right coroutine. The specific exceptions and error handling have to be added here.

This function is meant to be run as background task. Lock the handling of a resource with the `lock` attribute.

Parameters **run_once** (*bool, optional*) – if True, the function only handles one resource, then stops. Otherwise, continue to handle each new resource on the queue indefinitely.

list_app(*app*)
Accept the Applications that need to be managed by the Controller on listing them at startup. Starts the observer for the Applications with actual resources.

Parameters **app** (krake.data.kubernetes.Application) – the Application to accept or not.

on_status_update(*app*)
Called when an Observer noticed a difference of the status of an application. Request an update of the status on the API.

Parameters

- **app** (krake.data.kubernetes.Application) – the Application whose
- **has been updated or**(*status*) –

Returns the updated Application sent by the API.

Return type *krake.data.kubernetes.Application*

prepare(*client*)
Start all API clients that the controller will be using. Create all necessary coroutines and register them as background tasks that will be started by the Controller.

Parameters **client** (krake.client.Client) – the base client to use for the API client to connect to the API.

static scheduled_or_deleting(*app*)
Check if a resource should be accepted or not by the Controller to be handled.

Parameters **app** (krake.data.kubernetes.Application) – the Application to check.

Returns True if the Application should be handled, False otherwise.

Return type bool

class krake.controller.kubernetes.application.**KubernetesClient**(*kubeconfig*, *custom_resources=None*)

Bases: object

Client for connecting to a Kubernetes cluster. This client:

- prepares the connection based on the information stored in the cluster's kubeconfig file;
- prepares the connection to a custom resource's API, if a Kubernetes resource to be managed relies on a Kubernetes custom resource;
- offers two methods: - *apply()*: apply a manifest to create or update a resource - *delete()*: delete a resource.

The client can be used as a context manager, with the Kubernetes client being deleted when leaving the context.

kubeconfig
provided kubeconfig file, to connect to the cluster.

Type dict

custom_resources
name of all custom resources that are available on the current cluster.

Type list[str]

resource_apis
mapping of a Kubernetes's resource name to the API object of the Kubernetes client which manages it (e.g. a Pod belongs to the "CoreV1" API of Kubernetes, so the mapping would be "Pod" -> <client.CoreV1Api_instance>), wrapped in an `ApiAdapter` instance.

Type dict

api_client
base API object created by the Kubernetes API library.

Type *ApiClient*

apply(*resource*)
Apply the given resource on the cluster using its internal data as reference.

Parameters **resource** (*dict*) – the resource to create, as a manifest file translated in dict.

Returns response from the cluster as given by the Kubernetes client.

Return type object

custom_resource_apis
Determine custom resource apis for given cluster.

If given cluster supports custom resources, Krake determines apis from custom resource definitions.

The custom resources apis are requested only once and then are cached by cached property decorator. This is an advantage in case of the application contains multiple Kubernetes custom resources with the same kind, but with the different content, see example.

Example:

```
---
apiVersion: stable.example.com/v1
kind: CRD
metadata:
    name: cdr_1
spec:
    crdSpec: spec_1
---
apiVersion: stable.example.com/v1
kind: CRD
metadata:
    name: cdr_2
spec:
    crdSpec: spec_2
```

Returns Custom resource apis

Return type dict

Raises `InvalidCustomResourceDefinitionError` – If the request for the custom resource definition failed.

default_namespace
From the kubeconfig file, get the default Kubernetes namespace where the resources will be created. If no namespace is specified, "default" will be used.

Returns the default namespace in the kubeconfig file.

Return type str

delete(*resource*)

Delete the given resource on the cluster using its internal data as reference.

Parameters **resource** (*dict*) – the resource to delete, as a manifest file translated in dict.

Returns

response from the cluster as given by the Kubernetes client.

Return type kubernetes_asyncio.client.models.v1_status.V1Status

Raises

- InvalidManifestError – if the kind or name is not present in the resource.
- ApiException – by the Kubernetes API in case of malformed content or error on the cluster's side.

get_immutables(*resource*)

From a resource manifest, look for the group, version, kind, name and namespace of the resource.

If the latter is not present, the default namespace of the cluster is used instead.

Parameters **resource** (*dict[str, Any]*) – the manifest file translated in dict of the resource from which the fields will be extracted.

Returns

the group, version, kind, name and namespace of the resource.

Return type (str, str, str, str, str)

Raises InvalidResourceError – if the apiVersion, kind or the name is not present.

Raises

- InvalidManifestError – if the apiVersion, kind or name is not present in the resource.
- ApiException – by the Kubernetes API in case of malformed content or error on the cluster's side.

get_resource_api(*group*, *version*, *kind*)

Get the Kubernetes API corresponding to the given group and version. If not found, look for it into the supported custom resources for the cluster.

Parameters

- **group** (*str*) – group of the Kubernetes resource, for which the Kubernetes API should be retrieved.
- **version** (*str*) – version of the Kubernetes resource, for which the Kubernetes API should be retrieved.
- **kind** (*str*) – name of the Kubernetes resource, for which the Kubernetes API should be retrieved.

Returns the API adapter to use for this resource.

Return type ApiAdapter

Raises UnsupportedResourceError – if the group and version given are not supported by the Controller, and given kind is not a supported custom resource.

static log_response(*response*, *kind*, *action=None*)
Utility function to parse a response from the Kubernetes cluster and log its content.

Parameters

- **response** (*object*) – the response, as handed over by the Kubernetes client library.
- **kind** (*str*) – kind of the original resource that was managed (may be different from the kind of the response).
- **action** (*str*) – the type of action performed to get this response.

shutdown(*app*)
Gracefully shutdown the given application on the cluster by calling the apps exposed shutdown address.

Parameters **()** (*app*) – the app to gracefully shutdown.

Returns

response from the cluster as given by the Kubernetes client.

Return type kubernetes_asyncio.client.models.v1_status.V1Status

Raises

- InvalidManifestError – if the kind or name is not present in the resource.
- ApiException – by the Kubernetes API in case of malformed content or error on the cluster's side.

krake.controller.kubernetes.application.**register_service**(*app*, *cluster*, *resource*, *response*)
Register endpoint of Kubernetes Service object on creation and update.

Parameters

- **app** (krake.data.kubernetes.Application) – Application the service belongs to
- **cluster** (krake.data.kubernetes.Cluster) – The cluster on which the application is running
- **resource** (*dict*) – Kubernetes object description as specified in the specification of the application.
- **response** (*kubernetes_asyncio.client.V1Service*) – Response of the Kubernetes API

krake.controller.kubernetes.application.**unregister_service**(*app*, *resource*, ***kwargs*)
Unregister endpoint of Kubernetes Service object on deletion.

Parameters

- **app** (krake.data.kubernetes.Application) – Application the service belongs to
- **resource** (*dict*) – Kubernetes object description as specified in the specification of the application.

class krake.controller.kubernetes.application.**KubernetesApplicationObserver**(*cluster*, *resource*, *on_res_update*, *time_step=2*)
Bases: *krake.controller.Observer*

Observer specific for Kubernetes Applications. One observer is created for each Application managed by the Controller, but not one per Kubernetes resource (Deployment, Service...). If several resources are defined by an Application, they are all monitored by the same observer.

The observer gets the actual status of the resources on the cluster using the Kubernetes API, and compare it to the status stored in the API.

The observer is:

- started at initial Krake resource creation;
- deleted when a resource needs to be updated, then started again when it is done;
- simply deleted on resource deletion.

Parameters

- **cluster** (krake.data.kubernetes.Cluster) – the cluster on which the observed Application is created.
- **resource** (krake.data.kubernetes.Application) – the application that will be observed.
- **on_res_update** (*coroutine*) – a coroutine called when a resource's actual status differs from the status sent by the database. Its signature is: `(resource) -> updated_resource`. `updated_resource` is the instance of the resource that is up-to-date with the API. The Observer internal instance of the resource to observe will be updated. If the API cannot be contacted, `None` can be returned. In this case the internal instance of the Observer will not be updated.
- **time_step** (*int, optional*) – how frequently the Observer should watch the actual status of the resources.

poll_resource()

Fetch the current status of the Application monitored by the Observer.

Returns

the status object created using information from the real world Applications resource.

Return type *krake.data.core.Status*

krake.controller.kubernetes.application.**get_kubernetes_resource_idx**(*manifest*, *resource*, *check_namespace=False*)

Get a resource identified by its resource api, kind and name, from a manifest file

Parameters

- **manifest** (*list[dict]*) – Manifest file to get the resource from
- **resource** (*dict[str, dict|list|str]*) – resource to find
- **check_namespace** (*bool*) – Flag to decide, if the namespace should be checked

Raises IndexError – If the resource is not present in the manifest

Returns Position of the resource in the manifest

Return type int

class krake.controller.kubernetes.application.**HookType**

Bases: enum.Enum

An enumeration.

krake.controller.kubernetes.application.**update_last_applied_manifest_from_resp**(*app*, *response*, ***kwargs*)

Hook run after the creation or update of an application in order to update the *status.last_applied_manifest* using the k8s response.

Parameters

- **app** (krake.data.kubernetes.Application) – Application the service belongs to
- **response** (*kubernetes_asyncio.client.V1Status*) – Response of the Kubernetes API

After a Kubernetes resource has been created/updated, the *status.last_applied_manifest* has to be updated. All fields already initialized (either from the mangling of *spec.manifest*, or by a previous call to this function) should be left untouched. Only observed fields which are not present in *status.last_applied_manifest* should be initialized.

krake.controller.kubernetes.application.**update_last_observed_manifest_from_resp**(*app*, *response*, ***kwargs*)

Handler to run after the creation or update of a Kubernetes resource to update the last_observed_manifest from the response of the Kubernetes API.

Parameters

- **app** (krake.data.kubernetes.Application) – Application the service belongs to
- **response** (*kubernetes_asyncio.client.V1Service*) – Response of the Kubernetes API

The target last_observed_manifest holds the value of all observed fields plus the special control dictionaries for the list length

Controller Kubernetes Cluster

Module comprises Krake Kubernetes cluster controller logic.

class krake.controller.kubernetes.cluster.**KubernetesClusterController**(*api_endpoint*, *worker_count=10*, *loop=None*, *ssl_context=None*, *debounce=0*, *time_step=2*)

Bases: *krake.controller.Controller*

Controller responsible for *krake.data.kubernetes.Application* and *krake.data.kubernetes.Cluster* resources. The controller manages Application resources in "SCHEDULED" and "DELETING" state and Clusters in any state.

kubernetes_api
Krake internal API to connect to the "kubernetes" API of Krake.

Type *KubernetesApi*

cluster_reflector
reflector for the Cluster resource of the

Type *Reflector*

"kubernetes" API of Krake.

worker_count
the amount of worker function that should be run as background tasks.

Type int

observer_time_step
for the Observers: the number of seconds between two observations of the actual resource.

Type float

observers
mapping of all Application or Cluster resource' UID to their respective Observer and task responsible for the Observer. The signature is: `<uid> --> <observer>, <reference_to_observer's_task>`.

Type dict[str, (*Observer*, Coroutine)]

Parameters

- **api_endpoint** (*str*) – URL to the API
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.
- **ssl_context** (*ssl.SSLContext, optional*) – if given, this context will be used to communicate with the API endpoint.
- **debounce** (*float, optional*) – value of the debounce for the `WorkQueue`.
- **worker_count** (*int, optional*) – the amount of worker function that should be run as background tasks.
- **time_step** (*float, optional*) – for the Observers: the number of seconds between two observations of the actual resource.

static accept_accessible(*cluster*)
Check if a resource should be accepted or not by the Controller.

Parameters **cluster** (`krake.data.kubernetes.Cluster`) – the Cluster to check.

Returns True if the Cluster should be handled, False otherwise.

Return type bool

cleanup()
Unregister all background tasks that are attributes.

handle_resource(*run_once=False*)
Infinite loop which fetches and hand over the resources to the right coroutine. The specific exceptions and error handling have to be added here.

This function is meant to be run as background task. Lock the handling of a resource with the `lock` attribute.

Parameters **run_once** (*bool, optional*) – if True, the function only handles one resource, then stops. Otherwise, continue to handle each new resource on the queue indefinitely.

list_cluster(*cluster*)
Accept the Clusters that need to be managed by the Controller on listing them at startup. Starts the observer for the Cluster.

Parameters **cluster** (krake.data.kubernetes.Cluster) – the cluster to accept or not.

on_status_update(*cluster*)
Called when an Observer noticed a difference of the status of a resource. Request an update of the status on the API.

Parameters

- **cluster** (krake.data.kubernetes.Cluster) – the Cluster whose status
- **been updated.** (*has*) –

Returns the updated Cluster sent by the API.

Return type *krake.data.kubernetes.Cluster*

prepare(*client*)
Start all API clients that the controller will be using. Create all necessary coroutines and register them as background tasks that will be started by the Controller.

Parameters **client** (krake.client.Client) – the base client to use for the API client to connect to the API.

krake.controller.kubernetes.cluster.**register_service**(*app*, *cluster*, *resource*, *response*)
Register endpoint of Kubernetes Service object on creation and update.

Parameters

- **app** (krake.data.kubernetes.Application) – Application the service belongs to
- **cluster** (krake.data.kubernetes.Cluster) – The cluster on which the application is running
- **resource** (*dict*) – Kubernetes object description as specified in the specification of the application.
- **response** (*kubernetes_asyncio.client.V1Service*) – Response of the Kubernetes API

krake.controller.kubernetes.cluster.**unregister_service**(*app*, *resource*, ***kwargs*)
Unregister endpoint of Kubernetes Service object on deletion.

Parameters

- **app** (krake.data.kubernetes.Application) – Application the service belongs to
- **resource** (*dict*) – Kubernetes object description as specified in the specification of the application.

class krake.controller.kubernetes.cluster.**KubernetesClusterObserver**(*resource*, *on_res_update*, *time_step=2*)
Bases: *krake.controller.Observer*

Observer specific for Kubernetes Clusters. One observer is created for each Cluster managed by the Controller.

The observer gets the actual status of the cluster using the Kubernetes API, and compare it to the status stored in the API.

The observer is:

- started at initial Krake resource creation;
- deleted when a resource needs to be updated, then started again when it is done;
- simply deleted on resource deletion.

Parameters

- **resource** (krake.data.kubernetes.Cluster) – the cluster which will be observed.
- **on_res_update** (*coroutine*) – a coroutine called when a resource's actual status differs from the status sent by the database. Its signature is: `(resource) -> updated_resource`. `updated_resource` is the instance of the resource that is up-to-date with the API. The Observer internal instance of the resource to observe will be updated. If the API cannot be contacted, `None` can be returned. In this case the internal instance of the Observer will not be updated.
- **time_step** (*int, optional*) – how frequently the Observer should watch the actual status of the resources.

poll_resource()

Fetch the current status of the Cluster monitored by the Observer.

Note regarding exceptions handling: The current cluster status is fetched by *poll_resource()* from its API. If the cluster API is shutting down the API server responds with a 503 (service unavailable, apiserver is shutting down) HTTP response which leads to the kubernetes client ApiException. If the cluster's API has been successfully shut down and there is an attempt to fetch cluster status, the ClientConnectorError is raised instead. Therefore, both exceptions should be handled.

Returns

the status object created using information from the real world Cluster.

Return type *krake.data.core.Status*

class krake.controller.kubernetes.cluster.**HookType**

Bases: enum.Enum

An enumeration.

Controller Scheduler

Module comprises Krake scheduling logic of the Krake application.

class krake.controller.scheduler.**Scheduler**(*api_endpoint*, *worker_count=10*, *reschedule_after=60*, *stickiness=0.1*, *ssl_context=None*, *debounce=0*, *loop=None*)

Bases: *krake.controller.Controller*

The scheduler is a controller that receives all pending and updated applications and selects the "best" backend for each one of them based on metrics of the backends and application specifications.

Parameters

- **worker_count** (*int, optional*) – the amount of worker function that should be run as background tasks.
- **reschedule_after** (*float, optional*) – number of seconds after which a resource should be rescheduled.
- **ssl_context** (*ssl.SSLContext, optional*) – SSL context that should be used to communicate with the API server.
- **debounce** (*float, optional*) – number of seconds the scheduler should wait before it reacts to a state change.
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.

cleanup()
Unregister all background tasks that are attributes.

prepare(*client*)
Start all API clients that the controller will be using. Create all necessary coroutines and register them as background tasks that will be started by the Controller.

Parameters **client** (krake.client.Client) – the base client to use for the API client to connect to the API.

Controller Garbage Collector

This module defines the Garbage Collector present as a background task on the API application. When a resource is marked as deleted, the GC mark all its dependents as deleted. After cleanup is done by the respective Controller, the gc handles the final deletion of resources.

Marking a resource as deleted (by setting the deleted timestamp of its metadata) is irreversible: if the garbage collector receives such a resource, it will start the complete deletion process, with no further user involvement.

The configuration should have the following structure:

```
api_endpoint: http://localhost:8080
worker_count: 5
debounce: 1
tls:
  enabled: false
  client_ca: tmp/pki/ca.pem
  client_cert: tmp/pki/system:gc.pem
  client_key: tmp/pki/system:gc-key.pem

log:
  ...
```

exception krake.controller.gc.**DependencyCycleException**(*resource*, *cycle*, **args*)
Bases: *krake.controller.gc.DependencyException*

Raised in case a cycle in the dependencies has been discovered while adding or updating a resource.

Parameters

- **resource** (krake.data.core.ResourceRef) – the resource added or updated that triggered the exception.
- **cycle** (*set*) – the cycle of dependency relationships that has been discovered.

exception krake.controller.gc.**DependencyException**
Bases: Exception

Base class for dependency exceptions.

class krake.controller.gc.**DependencyGraph**
Bases: object

Representation of the dependencies of all Krake resources by an acyclic directed graph. This graph can be used to get the dependents of any resource that the graph received.

If an instance of a resource A depends on a resource B, A will have B in its owner list. In this case, * A depends on B * B is a dependency of A * A is a dependent of B

The nodes of the graph are *krake.data.core.ResourceRef*, created from the actual resources. The edges are directed links from a dependency to its dependents.

krake.data.core.ResourceRef are used instead of the resource directly, as they are hashable and can be used as key of a dictionary. Otherwise, we would need to make any newly added resource as hashable for the sake of the dependency graph.

The actual resources are still referenced in the `_resources`. It allows the access to the actual owners of a resource, not their *krake.data.core.ResourceRef*.

add_resource(*resource*, *owners*, *check_cycles=True*)
Add a resource and its dependencies relationships to the graph.

Parameters

- **resource** (krake.data.core.ResourceRef) – the resource to add to the graph.
- **owners** (*list*) – list of owners (dependencies) of the resource.
- **check_cycles** (*bool, optional*) – if False, does not check if adding the resource creates a cycle, and simply add it.

get_direct_dependents(*resource*)
Get the dependents of a resource, but only the ones directly dependent, no recursion is performed.

Parameters **resource** (krake.data.core.ResourceRef) – the resource for which the search will be performed.

Returns

the list of *krake.data.core.ResourceRef* to the dependents of the given resource (=that depends on the resource).

Return type list

remove_resource(*resource*, *check_dependents=True*)
If a resource has no dependent, remove it from the dependency graph, and from the dependents of other resources.

Parameters

- **resource** (krake.data.core.ResourceRef) – the resource to remove.
- **check_dependents** (*bool, optional*) – if False, does not check if the resource to remove has dependents, and simply remove it along with the dependents.

Raises *ResourceWithDependentsException* – if the resource to remove has dependents.

update_resource(*resource*, *owners*)
Update the dependency relationships of a resource on the graph.

Parameters

- **resource** (krake.data.core.ResourceRef) – the resource whose ownership may need to be modified.
- **owners** (*list*) – list of owners (dependencies) of the resource.

class krake.controller.gc.**GarbageCollector**(*api_endpoint*, *worker_count=10*, *loop=None*, *ssl_context=None*, *debounce=0*)

Bases: *krake.controller.Controller*

Controller responsible for marking the dependents of a resource as deleted, and for deleting all resources without any finalizer.

Parameters

- **api_endpoint** (*str*) – URL to the API
- **worker_count** (*int, optional*) – the amount of worker function that should be run as background tasks.
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.
- **ssl_context** (*ssl.SSLContext, optional*) – if given, this context will be used to communicate with the API endpoint.
- **debounce** (*float, optional*) – value of the debounce for the WorkQueue.

cleanup()

Unregister all background tasks that are attributes.

get_api_method(*reference*, *verb*)

Retrieve the client method of the API of the given resource to do the given action.

Parameters

- **reference** (*any*) – a resource or reference to a resource for which a method of its API needs to be selected.
- **verb** (*str*) – the verb describing the action for which the method should be returned.

Returns

a method to perform the given action on the given resource (through its client).

Return type callable

handle_resource(*run_once=False*)

Infinite loop which fetches and hand over the resources to the right coroutine. This function is meant to be run as background task.

Parameters **run_once** (*bool, optional*) – if True, the function only handles one resource, then stops. Otherwise, continue to handle each new resource on the queue indefinitely.

static is_in_deletion(*resource*)

Check if a resource needs to be deleted or not.

Parameters **resource** (krake.data.serializable.ApiObject) – the resource to check.

Returns True if the given resource is in deletion state, False otherwise.

Return type bool

on_received_deleted(*resource*)
To be called when a resource is deleted on the API. Remove the resource from the dependency graph and add its dependencies to the Worker queue.

Parameters **resource** (krake.data.serializable.ApiObject) – the deleted resource.

on_received_new(*resource*)
To be called when a resource is received for the first time by the garbage collector. Add the resource to the dependency graph and handle the resource if accepted.

If a cycle is detected when adding the resource, all resources of the cycle are removed.

Parameters **resource** (krake.data.serializable.ApiObject) – the newly added resource.

on_received_update(*resource*)
To be called when a resource is updated on the API. Update the resource on the dependency graph and handle the resource if accepted.

If a cycle is detected when adding the resource, all resources of the cycle are removed.

Parameters **resource** (krake.data.serializable.ApiObject) – the updated resource.

prepare(*client*)
Start all API clients that the controller will be using. Create all necessary coroutines and register them as background tasks that will be started by the Controller.

Parameters **client** (krake.client.Client) – the base client to use for the API client to connect to the API.

resource_received(*resource*)
Core functionality of the garbage collector. Mark the given resource's direct dependents as to be deleted, or remove the deletion finalizer if the resource has no dependent.

Parameters **resource** (krake.data.serializable.ApiObject) – a resource in deletion state.

exception krake.controller.gc.**ResourceWithDependentsException**(*dependents*, **args*)
Bases: *krake.controller.gc.DependencyException*

Raise when an attempt to remove a resource from the dependency graph implies removing a resource that has still dependents, and thus should not be removed if the integrity of the dependency graph needs to be kept.

For instance: If B depends on A, A should be removed.

Parameters **dependents** (*list*) – The list of dependents that are now orphaned.

Controller Magnum

Module for Krake controller responsible for managing Magnum cluster resources and creating their respective Kubernetes cluster. It connects to the Magnum service of the Project on which a MagnumCluster has been scheduled.

```
python -m krake.controller.magnum --help
```

Configuration is loaded from the `controllers.scheduler` section:

```
api_endpoint: http://localhost:8080
worker_count: 5
debounce: 1.0
poll_interval: 30

tls:
  enabled: false
  client_ca: tmp/pki/ca.pem
  client_cert: tmp/pki/system:magnum.pem
  client_key: tmp/pki/system:magnum-key.pem


log:
  ...
```

exception krake.controller.magnum.**CreateFailed**(*message*)
Bases: *krake.controller.ControllerError*

Raised in case the creation of a Magnum cluster failed.

exception krake.controller.magnum.**DeleteFailed**(*message*)
Bases: *krake.controller.ControllerError*

Raised in case the deletion of a Magnum cluster failed.

exception krake.controller.magnum.**InvalidClusterTemplateType**(*message*)
Bases: *krake.controller.ControllerError*

Raised in case the given Magnum template is not a template for a Kubernetes cluster.

class krake.controller.magnum.**MagnumClusterController**(**args*, *worker_count=5*, *poll_interval=30*, ***kwargs*)
Bases: *krake.controller.Controller*

The Magnum controller receives the MagnumCluster resources from the API and acts on it, by creating, updating or deleting their actual cluster counterparts. It uses the OpenStack Magnum client for this purpose.

Parameters

- **api_endpoint** (*str*) – URL to the API
- **loop** (*asyncio.AbstractEventLoop, optional*) – Event loop that should be used.
- **ssl_context** (*ssl.SSLContext, optional*) – if given, this context will be used to communicate with the API endpoint.
- **debounce** (*float, optional*) – value of the debounce for the `WorkQueue`.
- **worker_count** (*int, optional*) – the amount of worker function that should be run as background tasks.
- **poll_interval** (*float*) – time in second before two attempts to modify a Magnum cluster (creation, deletion, update, change from FAILED state...).

cleanup()
Unregister all background tasks that are attributes.

consume(*run_once=False*)
Continuously retrieve new elements from the worker queue to be processed.

Parameters **run_once** (*bool, optional*) – if True, the function only handles one resource, then stops. Otherwise, continue to handle each new resource on the queue indefinitely.

create_magnum_client(*cluster*)
Create a client to communicate with the Magnum service API for the given Magnum cluster. The specifications defined in the OpenStack project of the cluster are used to create the client.

Parameters **cluster** (*krake.data.openstack.MagnumCluster*) – the cluster whose project's specifications will be used to connect to the Magnum service.

Returns

the Magnum client to use to connect to the Magnum service on the project of the given Magnum cluster.

Return type MagnumV1Client

delete_magnum_cluster(*cluster*)
Initiate the deletion of the actual given Magnum cluster, and wait for its deletion. The finalizer specific to the Magnum Controller is also removed from the Magnum cluster resource.

Parameters **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster that needs to be deleted.

on_creating(*cluster*, *magnum*)
Called when a Magnum cluster with the CREATING state needs reconciliation.

Watch over a Magnum cluster currently being created on its scheduled OpenStack project, and updates the corresponding Kubernetes cluster created in the API.

As the Magnum cluster is in a stable state at the end, no further processing method is needed to return.

Parameters

- **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster that needs to be processed.
- **magnum** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service on the project.

on_pending(*cluster*, *magnum*)
Called when a Magnum cluster with the PENDING state needs reconciliation.

Initiate the creation of a Magnum cluster using the registered Magnum template, but does not ensure that the creation succeeded.

Parameters

- **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster to actually create on its scheduled OpenStack project.
- **magnum** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service on the project.

Returns

the next function to be called, as the Magnum cluster changed its state. In this case, the Magnum cluster has now the CREATING state, thus the function returned is *on_creating()*.

Return type callable

on_reconciling(*cluster*, *magnum*)
Called when a Magnum cluster with the RECONCILING state needs reconciliation.

Watch over a Magnum cluster already created on its scheduled OpenStack project, and updates the corresponding Kubernetes cluster created in the API.

As the Magnum cluster is in a stable state at the end, no further processing method is needed to return.

Parameters

- **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster that needs to be processed.
- **magnum** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service on the project.

on_running(*cluster*, *magnum*)
Called when a Magnum cluster with the RUNNING state needs reconciliation.

If the Magnum cluster needs to be resized, initiate the resizing. Otherwise, updates the corresponding Kubernetes cluster created in the API.

Parameters

- **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster that needs to be processed.
- **magnum** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service on the project.

Returns

the next function to be called, as the Magnum cluster changed its state. In the case of resizing, the Magnum cluster has now the RECONCILING state, thus the function returned is *on_creating()*. Otherwise, as the state is stable at the end, no further processing is needed and None is returned.

Return type callable

prepare(*client*)
Start all API clients that the controller will be using. Create all necessary coroutines and register them as background tasks that will be started by the Controller.

Parameters **client** (krake.client.Client) – the base client to use for the API client to connect to the API.

process_cluster(*cluster*)
Process a Magnum cluster: if the given cluster is marked for deletion, delete the actual cluster. Otherwise, start the reconciliation between a Magnum cluster spec and its state.

Handle any ControllerError or the supported OpenStack error that are raised during the processing.

Parameters **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster to process.

reconcile_kubernetes_resource(*cluster*, *magnum*)
Create or update the Krake resource of the Kubernetes cluster that was created from a given Magnum cluster.

Parameters

- **cluster** (*krake.data.openstack.MagnumCluster*) – the Kubernetes cluster will be created using the specifications of this Magnum cluster.

- **magnum** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service on the project.

Raises ClientResponseError – when checking if the Kubernetes cluster resource already exists, raise if any HTTP error except 404 is raised.

reconcile_magnum_cluster(*cluster*)

Depending on the state of the given Magnum cluster, start the rapprochement of the wanted state of the cluster to the desired one.

Parameters **cluster** (*krake.data.openstack.MagnumCluster*) – the cluster whose actual state will be modified to match the desired one.

wait_for_running(*cluster*, *magnum*)

Await for an actual Magnum cluster to be in a stable state, that means, when its creation or update is finished.

Parameters

- **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster on which an operation is performed that needs to be awaited.
- **magnum** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service on the project.

Raises ControllerError – if the operation on the cluster failed, a corresponding error will be raised (for instance CreateFailed in case the creation of the cluster failed).

exception krake.controller.magnum.**ReconcileFailed**(*message*)

Bases: *krake.controller.ControllerError*

Raised in case the update of a Magnum cluster failed.

krake.controller.magnum.**concurrent**(*fn*)

Decorator function to turn a synchronous function into an asynchronous coroutine that runs in another thread, that can be awaited and thus does not block the main asyncio loop. It is particularly useful for synchronous tasks which requires a long time to be run concurrently to the main asyncio loop.

Example

```
@concurrent
def my_function(args_1, arg2=value):
    # long synchronous processing...
    return result

await my_function(value1, arg2=value2)  # function run in another thread
```

Parameters **fn** (*callable*) – the function to run in parallel from the main loop.

Returns

decorator around the given function. The returned callable is an asyncio coroutine.

Return type callable

krake.controller.magnum.**create_client_certificate**(*client*, *cluster*, *csr*)

Create and get a certificate for the given Magnum cluster.

Parameters

- **client** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service.
- **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster for which a kubeconfig file will be created.
- **csr** (*str*) – the certificate signing request (CSR) to use on the Magnum service for the creation of the certificate.

Returns the generated certificate.

Return type str

krake.controller.magnum.**create_magnum_cluster**(*client*, *cluster*)

Create an actual Magnum cluster by connecting to the the Magnum service.

Parameters

- **client** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service.
- **cluster** (*krake.data.openstack.MagnumCluster*) – the cluster to create.

Returns the cluster created by the Magnum service.

Return type magnumclient.v1.clusters.Cluster

krake.controller.magnum.**delete_magnum_cluster**(*client*, *cluster*)

Delete the actual Magnum cluster that corresponds to the given resource.

Parameters

- **client** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service.
- **cluster** (*krake.data.openstack.MagnumCluster*) – the cluster to delete.

Returns the cluster deleted by the Magnum service.

Return type magnumclient.v1.clusters.Cluster

krake.controller.magnum.**format_openstack_error**(*error*)

Create a more readable error message using OpenStack specific errors.

Parameters **error** (*BaseException*) – the exception whose information is used to create a message.

Returns the generated error message.

Return type str

krake.controller.magnum.**generate_magnum_cluster_name**(*cluster*)

Create a unique name for a Magnum cluster from its metadata. The name has the following structure: "<namespace>-<name>-<random_lowercase_digit_string>". Any special character that the Magnum service would see as invalid will be replaced.

Parameters **cluster** (*krake.data.openstack.MagnumCluster*) – the cluster to use to create a name.

Returns the name generated.

Return type str

krake.controller.magnum.**make_csr**(*key_size=4096*)

Generates a private key and corresponding certificate and certificate signing request.

Parameters **key_size** (*int*) – Length of private key in bits

Returns private key, certificate signing request (CSR)

Return type (str, str)

krake.controller.magnum.**make_keystone_session**(*project*)
Create an OpenStack Keystone session using the authentication information of the given project resource.

Parameters **project** (*krake.data.openstack.Project*) – the OpenStack project to use for getting the credentials and endpoint.

Returns the Keystone session created.

Return type *Session*

krake.controller.magnum.**make_kubeconfig**(*client*, *cluster*)
Create a kubeconfig for the Kubernetes cluster associated with the given Magnum cluster. For this process, it uses (non exhaustively) the name, address and certificates associated with it.

Parameters

- **client** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service.
- **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster for which a kubeconfig will be created.

Returns the kubeconfig created, returned as a dictionary.

Return type dict

krake.controller.magnum.**make_magnum_client**(*project*)
Create a Magnum client to connect to the given OpenStack project.

Parameters **project** (*krake.data.openstack.Project*) – the project to connect to.

Returns

the client to connect to the Magnum service of the given project.

Return type MagnumV1Client

krake.controller.magnum.**randstr**(*length=7*)
Create a random string of lowercase and digit character of the given length.

Parameters **length** (*int*) – specifies how many characters should be present in the returned string.

Returns the string randomly generated.

Return type str

krake.controller.magnum.**read_ca_certificate**(*client*, *cluster*)
Get the certificate authority used by the given Magnum cluster.

Parameters

- **client** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service.
- **cluster** (*krake.data.openstack.MagnumCluster*) – the Magnum cluster for which the certificate authority will be retrieved.

Returns the certificate authority of the given cluster.

Return type str

krake.controller.magnum.**read_magnum_cluster**(*client*, *cluster*)
Read the actual information of the given Magnum cluster resource.

Parameters

- **client** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service.
- **cluster** (*krake.data.openstack.MagnumCluster*) – the resource whose actual cluster state will be read.

Returns

the current information regarding the given Magnum cluster.

Return type magnumclient.v1.clusters.Cluster

krake.controller.magnum.**read_magnum_cluster_template**(*client*, *cluster*)
Get the actual template associated with the one specified in the given Magnum cluster resource.

Parameters

- **client** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service.
- **cluster** (*krake.data.openstack.MagnumCluster*) – the template given is the one specified by this Magnum cluster.

Returns magnumclient.v1.cluster_templates.ClusterTemplate

krake.controller.magnum.**resize_magnum_cluster**(*client*, *cluster*)
Update the given Magnum cluster by changing its node count.

Parameters

- **client** (*MagnumV1Client*) – the Magnum client to use to connect to the Magnum service.
- **cluster** (*krake.data.openstack.MagnumCluster*) – the cluster to resize.

Returns the cluster updated by the Magnum service.

Return type magnumclient.v1.clusters.Cluster

4.15.6 Data Abstraction

Data abstraction module for all REST resources used by the Krake API. This module provides common data definitions for *krake.api* and *krake.client*.

The core functionality is provided by *serializable* providing a Python API for declarative definitions of data models together with serializing and deserializing functionality.

Domain-specific models are defined in corresponding submodules, e.g. Kubernetes-related data models are defined in *kubernetes*.

class krake.data.**Key**(*template*, *attribute=None*)
Bases: object

Etcd key template using the same syntax as Python's standard format strings for parameters.

Example

```
key = Key("/books/{namespaces}/{isbn}")
```

The parameters are substituted by in the corresponding methods by either attributes of the passed object or additional keyword arguments.

Parameters

- **template** (*str*) – Key template with format string-like parameters
- **attribute** (*str, optional*) – Load attributes in *format_object()* from this attribute of the passed object.

format_kwargs(***kwargs*)

Create a key from keyword arguments

Parameters ****kwargs** – Keyword arguments for parameter substitution

Returns Key from the key template with all parameters substituted by the given keyword arguments.

Return type str

format_object(*obj*)

Create a key from a given object

If `attribute` is given, attributes are loaded from this attribute of the object rather than the object itself.

Parameters **obj** (*object*) – Object from which attributes are looked up

Returns Key from the key template with all parameters substituted by attributes loaded from the given object.

Return type str

Raises `AttributeError` – If a required parameter is missing

matches(*key*)

Check if a given key matches the template

Parameters **key** (*str*) – Key that should be checked

Returns True of the given key matches the key template

Return type bool

prefix(***kwargs*)

Create a partial key (prefix) for a given object.

Parameters ****kwargs** – Parameters that will be used for substitution

Returns Partial key from the key template with some parameters substituted

Return type str

Raises `TypeError` – If a parameter is passed as keyword argument but a preceding parameter is not given.

`krake.data.`**persistent**(*key*)

Decorator factory for marking a class with a template that should be used as etcd key.

The passed template will be converted into a *Key* instance using the `metadata` attribute and will be assigned to the `__etcd_key__` attribute of the decorated class.

Example

```
from krake.data import persistent
from krake.data.serializable import Serializable, persistent
from krake.data.core import Metadata

@persistent("/books/{name}")
class Book(Serializable):
    metadata: Metadata
```

Parameters **key** (*str*) – Etcd key template. Parameters will be loaded from the `metadata` attribute of the decorated class.

Returns Decorator that can be used to assign an `__etcd_key__` attribute to the decorated object based on the passed key template.

Return type callable

This module defines a declarative API for defining data models that are JSON-serializable and JSON-deserializable.

class krake.data.serializable.**ApiObject**(***kwargs*)

Bases: *krake.data.serializable.Serializable*

Base class for objects manipulated via REST API.

api and *kind* should be defined as simple string class :variables. They are automatically converted into dataclass fields with :corresponding validators.

api

Name of the API the object belongs to

Type str

kind

String name describing the kind (type) of the object

Type str

Example

```
from krake.data.serializable import ApiObject
from krake.data.core import Metadata, Status

class Book(ApiObject):
    api: str = "shelf"  # The book resource belongs to the "shelf api"
    kind: str = "Book"

    metadata: Metadata
    spec: BookSpec
    status: Status
```

class **Schema**(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)

Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.serializable.**ModelizedSchema**(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: marshmallow.schema.Schema

Simple marshmallow schema constructing Python objects in a `post_load` hook.

Subclasses can specify a callable attribute `__model__` which is called with all deserialized attributes as keyword arguments.

The `Meta.unknown` field is set to avoid considering unknown fields during validation. It mostly prevents create tests from failing.

__model__
Model factory returning a new instance of a specific model

Type callable

class krake.data.serializable.**PolymorphicContainer**(***kwargs*)

Bases: *krake.data.serializable.Serializable*

Base class for polymorphic serializable objects.

The polymorphic serializable has a string attribute `type` which is used as discriminator for the different types. There is an attribute named exactly like the value of the `type` attribute containing the deserialized subtype.

Every new subclass will create its own *Schema* attribute. This means every subclass has its own internal subtype registry.

Schema
Schema that will be used for (de-)serialization of the class.

Type *PolymorphicContainerSchema*

Example:

```
from krake.data.serializable import Serializable, PolymorphicContainer

class ValueSpec(PolymorphicContainer):
    pass

@ProviderSpec.register("float")
class FloatSpec(Serializable):
    min: float
    max: float

@ProviderSpec.register("bool")
class BoolSpec(Serializable):
    pass

# Deserialization
spec = ProviderSpec.deserialize({
    "type": "float",
    "float": {
        "min": 0,
        "max": 1.0,
```

(continues on next page)

(continued from previous page)

```
    },
})
assert isinstance(spec.float, FloatSpec)

# Serialization
assert ProviderSpec(type="bool", bool=BoolSpec()).serialize() == {
    "type": bool,
    "bool": {},
}
```

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *krake.data.serializable.ModelizedSchema*

classmethod register(*name*)

Decorator function for registering a class under a unique name.

Parameters **name** (*str*) – Name that will be used as value for the `type` field to identify the decorated class.

Returns Decorator that will register the decorated class in the polymorphic schema (see *PolymorphicContainerSchema.register()*).

Return type callable

update(*overwrite*)

Update the polymorphic container with fields from the overwrite object.

A reference to the polymorphic field – the field called like the value of the `type` attribute – of the overwrite object is assigned to the current object even if the types of the current object and the overwrite object are identical.

Parameters **overwrite** (Serializable) – Serializable object will be merged with the current object.

class krake.data.serializable.**PolymorphicContainerSchema**(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: marshmallow.schema.Schema

Schema that is used by *PolymorphicContainer*

It declares just one string field `type` which is used as discriminator for the different types.

There should be a field called exactly like the type. The value of this field is passed to the registered schema for deserialization.

```
---
type: float
float:
    min: 0
    max: 1.0
---
type: int
int:
    min: 0
    max: 100
```

Every subclass will create its own internal subtype registry.

classmethod register(*type*, *dataclass*)
Register a *Serializable* for the given type string

Parameters

- **type** (*str*) – Type name that should be used as discriminator
- **dataclass** (*object*) – Dataclass that will be used when the type field equals the specified name.

Raises ValueError – If the type name is already registered

class krake.data.serializable.**Serializable**(***kwargs*)
Bases: object

Base class for declarative serialization API.

Fields can be marked with the metadata attribute of dataclasses.Field. Currently the following markers exists:

readonly A field marked as "readonly" is automatically generated by the API server and not controlled by the user. The user cannot update this field. The corresponding marshmallow field allows None as valid value.

subresource A field marked as "subresource" is ignored in update request of a resource. Extra REST call are required to update a subresource. A well known subresource is "status".

All field metadata attributes are also passed to the marshmallow.fields.Field instance. This means the user can control the generated marshmallow field with the metadata attributes.

The class also defines a custom __init__ method accepting every attribute as keyword argument in arbitrary order in contrast to the standard init method of dataclasses.

Example

```
from krake.data.serializable import Serializable

class Book(Serializable):
    author: str
    title: str
    isbn: str = fields(metadata={"readonly": True})

assert hasattr(Book, "Schema")
```

There are cases where multiple levels needs to be validated together. In this case, the validates metadata key for a single field is not sufficient anymore. One solution is to overwrite the auto-generated schema by a custom schema using the marshmallow.decorators.validates_schema() decorator.

Another solution is leveraging the `__post_init__()` method of dataclasses. The fields can be validated in this method and a raised `marshmallow.ValidationError` will propagate to the Schema deserialization method.

```
from marshmallow import ValidationError

class Interval(Serializable):
    max: int
    min: int

    def __post_init__(self):
        if self.min > self.max:
            raise ValidationError("'min' must not be greater than 'max'")

# This will raise a ValidationError
interval = Interval.deserialize({"min": 2, "max": 1})
```

Schema
Schema for this dataclass

Type *ModelizedSchema*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

__post_init__()
The `__init__()` method calls this method after all fields are initialized.

It is mostly useful for schema-level validation (see above).

For now, *Serializable* does not support init-only variables because they do not make much sense for object stored in a database. This means no additional parameters are passed to this method.

classmethod deserialize(*data*, *creation_ignored=False*)
Loading an instance of the class from JSON-encoded data.

Parameters

- **data** (*dict*) – JSON dictionary that should be deserialized.
- **creation_ignored** (*bool*) – if True, all attributes not needed at the creation are ignored. This contains the read-only and subresources, which can only be created by the API.

Raises `marshmallow.ValidationError` – If the data is invalid

classmethod fields_ignored_by_creation()
Return the name of all fields that do not have to be provided during the creation of an instance.

Returns Set of name of fields that are either subresources or read-only, or nested read-only fields.

Return type set

classmethod readonly_fields(*prefix=None*)
Return the name of all read-only fields. Nested fields are returned with dot-notation, for lists also. In this case, the argument is the one taken into account for looking at the read-only fields.

Example:

```
class Comment(Serializable):
    id: int = field(metadata={"readonly": True})
    content: str

class BookMetadata(Serializable):
    name: str = field(metadata={"readonly": True})
    published: datetime = field(metadata={"readonly": True})
    last_borrowed: datetime

class Book(Serializable):
    id: int = field(metadata={"readonly": True})
    metadata: BookMetadata
    status: str
    comments: List[Comment]

expected = {'id', 'metadata.name', 'metadata.published', 'comment.id'}
assert Book.readonly_fields() == expected
```

Parameters **prefix** (*str, optional*) – Used for internal recursion

Returns Set of field names that are marked as with `readonly` in their metadata.

Return type set

serialize(*creation_ignored=False*)

Serialize the object using the generated *Schema*.

Parameters **creation_ignored** (*bool*) – if True, all attributes not needed at the creation are ignored. This contains the read-only and subresources, which can only be created by the API.

Returns JSON representation of the object

Return type dict

classmethod subresources_fields()

Return the name of all fields that are defined as subresource.

Returns

Set of field names that are marked as subresource in their metadata

Return type set

update(*overwrite*)

Update data class fields with corresponding fields from the overwrite object.

If a field is marked as _subresource_ or _readonly_ it is not modified. If a field is marked as _immutable_ and there is an attempt to update the value, the `ValueError` is raised. Otherwise, attributes from overwrite will replace attributes from the current object.

The *update()* must ignore the _subresource_ and _readonly_ fields, to avoid accidentally overwriting e.g. status fields in read-modify-write scenarios.

The function works recursively for nested *Serializable* attributes which means the *update()* method of the attribute will be used. This means the identity of a *Serializable* attribute will not change unless the current attribute or the overwrite attribute is `None`.

All other attributes are updated by assigning **references** from the overwrite attributes to the current object. This leads to a behavior similar to "shallow copying" (see `copy.copy()`). If the attribute is mutable,

e.g. `list` or `dict`, the attribute in the current object will reference the same object as in the overwrite object.

Parameters **overwrite** (`Serializable`) – Serializable object will be merged with the current object.

Raises `ValueError` – If there is an attempt to update an _immutable_ field.

class krake.data.serializable.**SerializableMeta**

Bases: `type`

Metaclass for *Serializable*. It automatically converts a specified class into an dataclass (see `dataclasses.dataclass()`) and creates a corresponding `marshmallow.Schema` class. The schema class is assigned to the `Schema` attribute.

krake.data.serializable.**field_for_schema**(*type_*, *default=<dataclasses._MISSING_TYPE object>*, ***metadata*)

Create a corresponding `marshmallow.fields.Field` for the passed type.

If `metadata` contains `marshmallow_field` key, the value will be used directly as field.

If `type_` has a `Schema` attribute which should be a subclass of `marshmallow.Schema` a :class.'marshmallow.fields.Nested' field will be returned wrapping the schema.

If `type_` has a `Field` attribute which should be a subclass of `marshmallow.fields.Field` an instance of this attribute will be returned.

Parameters

- **type** (*type*) – Type of the field
- **default** (*optional*) – Default value of the field
- ****metadata** (*dict*) – Any additional keyword argument that will be passed to the field

Returns Serialization field for the passed type

Return type marshmallow.fields.Field

Raises `NotImplementedError` – If the marshmallow field cannot not be determined for the passed type

krake.data.serializable.**is_base_generic**(*cls*)

Detects generic base classes, for example `List` but not `List[int]`.

Parameters **cls** – Type annotation that should be checked

Returns True if the passed type annotation is a generic base.

Return type bool

krake.data.serializable.**is_generic**(*cls*)

Detects any kind of generic, for example *List* or *List[int]*. This includes "special" types like Union and Tuple - anything that's subscriptable, basically.

Parameters **cls** – Type annotation that should be checked

Returns True if the passed type annotation is a generic.

Return type bool

krake.data.serializable.**is_generic_subtype**(*cls*, *base*)

Check if a given generic class is a subtype of another generic class

If the base is a qualified generic, e.g. `List[int]`, it is checked if the types are equal. If the base or cls does not have the attribute *__origin__*, e.g. Union, Optional, it is checked, if the type of base or cls is equal to the

opponent. This is done for every possible case. If the base and cls have the attribute *__origin__*, e.g. `list` for `typing.List`, it is checked if the class is equal to the original type of the generic base class.

Parameters

- **cls** – Generic type
- **base** – Generic type that should be the base of the given generic type.

Returns True of the given generic type is a subtype of the given base generic type.

Return type bool

krake.data.serializable.**is_qualified_generic**(*cls*)

Detects generics with arguments, for example `List[int]` but not `List`

Parameters **cls** – Type annotation that should be checked

Returns True if the passed type annotation is a qualified generic.

Return type bool

.

class krake.data.core.**BaseMetric**(***kwargs*)

Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)

Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**BaseMetricsProvider**(***kwargs*)

Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)

Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**Conflict**(***kwargs*)

Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)

Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**CoreMetadata**(***kwargs*)

Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)

Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**GlobalMetric**(***kwargs*)

Bases: *krake.data.core.BaseMetric*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**GlobalMetricList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**GlobalMetricsProvider**(***kwargs*)
Bases: *krake.data.core.BaseMetricsProvider*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**GlobalMetricsProviderList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**KafkaSpec**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Specifications to connect to a KSQL database, and retrieve a specific row from a specific table.

comparison_column
name of the column where the value will be compared to the metric name, to select the right metric.

Type str

value_column
name of the column where the value of a metric is stored.

Type str

table
the name of the KSQL table where the metric is defined.

Type str

url
endpoint of the KSQL database.

Type str

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**ListMetadata**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**Metadata**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**Metric**(***kwargs*)
Bases: *krake.data.core.BaseMetric*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**MetricList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**MetricRef**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**MetricSpec**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**MetricSpecProvider**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**MetricsProvider**(***kwargs*)
Bases: *krake.data.core.BaseMetricsProvider*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**MetricsProviderList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**MetricsProviderSpec**(***kwargs*)
Bases: *krake.data.serializable.PolymorphicContainer*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.PolymorphicContainerSchema*

class krake.data.core.**PrometheusSpec**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**Reason**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**ReasonCode**
Bases: enum.IntEnum

An enumeration.

class krake.data.core.**ResourceRef**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**Role**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**RoleBinding**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**RoleBindingList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**RoleList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**RoleRule**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**StaticSpec**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**Status**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**Verb**
Bases: enum.Enum

An enumeration.

class krake.data.core.**WatchEvent**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.core.**WatchEventType**
Bases: enum.Enum

An enumeration.

krake.data.core.**resource_ref**(*resource*)
Create a *ResourceRef* from a ApiObject

Parameters **resource** (serializable.ApiObject) – API object that should be referenced

Returns Corresponding reference to the API object

Return type *ResourceRef*

krake.data.core.**validate_key**(*key*)
Validate the given key against the corresponding regular expression.

Parameters **key** – the string to validate

Raises ValidationError – if the given key is not conform to the regular expression.

krake.data.core.**validate_value**(*value*)
Validate the given value against the corresponding regular expression.

Parameters **value** – the string to validate

Raises ValidationError – if the given value is not conform to the regular expression.

class krake.data.infrastructure.**Cloud**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**CloudBinding**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**CloudList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**CloudSpec**(***kwargs*)
Bases: *krake.data.serializable.PolymorphicContainer*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.PolymorphicContainerSchema*

class krake.data.infrastructure.**CloudState**
Bases: enum.Enum

An enumeration.

class krake.data.infrastructure.**CloudStatus**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Status subresource of *GlobalCloud* and *Cloud*.

state
Current state of the cloud.

Type *CloudState*

metrics_reasons
Mapping of the name of the metrics for which an error occurred to the reason for which it occurred.

Type dict[str, *Reason*]

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**GlobalCloud**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

__post_init__()
Method automatically ran at the end of the __init__() method, used to validate dependent attributes.

Validations:

1. A non-namespaced *GlobalCloud* resource cannot reference the namespaced

   InfrastructureProvider resource, see #499 for details

2. A non-namespaced *GlobalCloud* resource cannot reference the namespaced

   Metric resource, see #499 for details

Note: This validation cannot be achieved directly using the `validate` metadata, since `validate` must be a zero-argument callable, with no access to the other attributes of the dataclass.

class krake.data.infrastructure.**GlobalCloudList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**GlobalInfrastructureProvider**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**GlobalInfrastructureProviderList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**ImSpec**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

IMSpec should contain access data to the IM provider instance.

url
endpoint of the IM provider instance.

Type str

username
IM provider instance username.

Type str, optional

password
IM provider instance password.

Type str, optional

token
IM provider instance token.

Type str, optional

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

__post_init__()
Method automatically ran at the end of the `__init__()` method, used to validate dependent attributes.

Validations: - At least one of the attributes from the following should be defined:

- *username* and *password*
- *token*

class krake.data.infrastructure.**InfrastructureProvider**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**InfrastructureProviderList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**InfrastructureProviderRef**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**InfrastructureProviderSpec**(***kwargs*)
Bases: *krake.data.serializable.PolymorphicContainer*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.PolymorphicContainerSchema*

class krake.data.infrastructure.**OpenstackAuthMethod**(***kwargs*)
Bases: *krake.data.serializable.PolymorphicContainer*

Container for the different authentication strategies of OpenStack Identity service (Keystone).

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.PolymorphicContainerSchema*

class krake.data.infrastructure.**OpenstackSpec**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**Password**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Data for the password authentication strategy of the OpenStack identity service (Keystone).

version
OpenStack identity API version used for authentication

Type str

user
OpenStack user that will be used for authentication

Type *UserReference*

project
OpenStack project that will be used by Krake

Type *ProjectReference*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**ProjectReference**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Reference to the OpenStack project that is used by the *Password* authentication strategy.

name
Name or UUID of the OpenStack project

Type str

domain_id
Domain ID of the OpenStack project. Defaults to *default*

Type str, optional

comment
Arbitrary string for user-defined information, e.g. semantic names

Type str, optional

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.infrastructure.**UserReference**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Reference to the OpenStack user that is used by the *Password* authentication strategy.

username
Username or UUID of the OpenStack user

Type str

password
Password of the OpenStack user

Type str

domain_name
Domain name of the OpenStack user. Defaults to *Default*

Type str, optional

comment
Arbitrary string for user-defined information, e.g. semantic names

Type str, optional

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

Data model definitions for Kubernetes-related resources

class krake.data.kubernetes.**Application**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ApplicationComplete**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ApplicationList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ApplicationShutdown**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ApplicationSpec**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Spec subresource of *Application*.

manifest
List of Kubernetes resources to create. This attribute is managed by the user.

Type list[dict]

tosca
The to be created TOSCA template. A TOSCA template should be defined as a python dict or with the URL, where the template is located. This attribute is managed by the user.

Type Union[dict, str], optional

csar
The to be created CSAR archive. A CSAR file should be defined with the URL, where the archive is located. This attribute is managed by the user.

Type str, optional

observer_schema
List of dictionaries of fields that should be observed by the Kubernetes Observer. This attribute is managed by the user. Using this attribute as a basis, the Kubernetes Controller generates the `status.mangled_observer_schema`.

Type list[dict], optional

constraints
Scheduling constraints

Type *Constraints*, optional

hooks
List of enabled hooks

Type list[str], optional

shutdown_grace_time
timeout in seconds for the shutdown hook

Type int

backoff
multiplier applied to backoff_delay between attempts. default: 1 (no backoff)

Type field, optional

backoff_delay
delay [s] between attempts. default: 1

Type field, optional

backoff_limit
a maximal number of attempts, default: -1 (infinite)

Type field, optional

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

__post_init__()
Method automatically ran at the end of the `__init__()` method, used to validate dependent attributes.

Validations: 1. At least one of the attributes from the following should be defined: - *manifest* - *tosca* - *csar* If the user specified multiple attributes at once, the *manifest* has the highest priority, after that *tosca* and *csar*.

2. If a custom *observer_schema* and *manifest* are specified by the user, the *observer_schema* needs to be validated, i.e. verified that resources are correctly identified and refer to resources defined in *manifest*, that fields are correctly identified and that all special control dictionaries are correctly defined.

Note: These validations cannot be achieved directly using the `validate` metadata, since `validate` must be a zero-argument callable, with no access to the other attributes of the dataclass.

class krake.data.kubernetes.**ApplicationState**
Bases: enum.Enum

An enumeration.

class krake.data.kubernetes.**ApplicationStatus**(***kwargs*)
Bases: *krake.data.core.Status*

Status subresource of *Application*.

state
Current state of the application

Type *ApplicationState*

container_health
Specific details of the application

Type *ContainerHealth*

kube_controller_triggered
Timestamp that represents the last time the current version of the Application was scheduled (version here meaning the Application after an update). It is only updated after the update of the Application led to a rescheduling, or at the first scheduling. It is used to keep a strict workflow between the Scheduler and Kubernetes Controller: the first one should always handle an Application creation or update before the latter. Only after this field has been updated by the Scheduler to be higher than the modified timestamp can the Kubernetes Controller handle the Application.

Type datetime.datetime

scheduled
Timestamp that represents the last time the application was scheduled to a different cluster, in other words when `scheduled_to` was modified. Thus, it is updated at the first binding to a cluster, or during the binding with a different cluster. This represents the timestamp when the current Application was scheduled to its current cluster, even if it has been updated in the meantime.

Type datetime.datetime

scheduled_to
Reference to the cluster where the application should run.

Type *ResourceRef*

running_on
Reference to the cluster where the application is currently running.

Type *ResourceRef*

services
Mapping of Kubernetes service names to their public endpoints.

Type dict

mangled_observer_schema
Actual observer schema used by the Kubernetes Observer, generated from the user inputs `spec.observer_schema`

Type list[dict]

last_observed_manifest
List of Kubernetes resources observed on the Kubernetes API.

Type list[dict]

last_applied_manifest
List of Kubernetes resources created via Krake. The manifest is augmented by additional resources needed to be created for the functioning of internal mechanisms, such as the "Complete Hook".

Type list[dict]

complete_token
Token to identify the "Complete Hook" request

Type str

complete_cert
certificate for the identification of the "Complete Hook".

Type str

complete_key
key for the certificate of the "Complete Hook" identification.

Type str

shutdown_token
Token to identify the "Shutdown Hook" request

Type str

shutdown_cert
certificate for the identification of the "Shutdown Hook".

Type str

shutdown_key
key for the certificate of the "Shutdown Hook" identification.

Type str

shutdown_grace_period
time period the shutdown method waits on after the shutdown command was issued to an object

Type datetime

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**CloudConstraints**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Constraints for the `Cloud` to which this cluster is scheduled.

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**Cluster**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterBinding**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterCloudConstraints**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Constraints restricting the scheduling decision for a *Cluster*.

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterConstraints**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterList**(***kwargs*)
Bases: *krake.data.serializable.ApiObject*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterNode**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Cluster node subresource of *ClusterStatus*.

api
Api version if the resource.

Type str, optional

kind
Kind of the resource.

Type str, optional

status
Current status of the cluster node.

Type *ClusterNodeStatus*, optional

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterNodeCondition**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Cluster node condition subresource of *ClusterNodeStatus*.

message
Human readable message indicating details about last transition.

Type str

reason
A brief reason for the condition's last transition.

Type str

status
Status of the condition, one of "True", "False", "Unknown".

Type str

type
Type of node condition.

Type str

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterNodeMetadata**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Cluster node metadata subresource of *ClusterNode*.

name
Name of the cluster node.

Type str

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterNodeStatus**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

Cluster node status subresource of *ClusterNode*.

conditions
List of current observed node conditions.

Type list[*ClusterNodeCondition*]

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ClusterSpec**(**kwargs*)
Bases: *krake.data.serializable.Serializable*

Spec subresource of *Cluster*

kubeconfig
path to the kubeconfig file for the cluster to register.

Type dict

custom_resources
name of all custom resources that are available on the current cluster.

Type list

metrics
metrics used on the cluster.

Type list

backoff
multiplier applied to backoff_delay between attempts. default: 1 (no backoff)

Type field, optional

backoff_delay
delay [s] between attempts. default: 1

Type field, optional

backoff_limit
a maximal number of attempts, default: -1 (infinite)

Type field, optional

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

__post_init__()
Method automatically ran at the end of the __init__() method, used to validate dependent attributes.

Validations: - At least one of the attributes from the following should be defined:

- *kubeconfig*
- tosca

Note: This validation cannot be achieved directly using the validate metadata, since validate must be a zero-argument callable, with no access to the other attributes of the dataclass.

class krake.data.kubernetes.**ClusterState**
Bases: enum.Enum

An enumeration.

class krake.data.kubernetes.**ClusterStatus**(***kwargs*)
Bases: *krake.data.core.Status*

Status subresource of *Cluster*.

kube_controller_triggered
Time when the Kubernetes controller was

Type datetime

triggered. This is used to handle cluster state transitions.

state
Current state of the cluster.

Type *ClusterState*

metrics_reasons
mapping of the name of the metrics for which an error occurred to the reason for which it occurred.

Type dict[str, *Reason*]

last_applied_tosca
TOSCA template applied via Krake.

Type dict

nodes
list of cluster nodes.

Type list[*ClusterNode*]

cluster_id
UUID or name of the cluster (infrastructure) given by the infrastructure provider

Type str

scheduled
Timestamp that represents the last time the cluster was scheduled to a cloud.

Type datetime.datetime

scheduled_to
Reference to the cloud where the cluster should run.

Type *ResourceRef*

running_on
Reference to the cloud where the cluster is running.

Type *ResourceRef*

retries
Count of remaining retries to access the cluster. Is set via the Attribute backoff in in ClusterSpec.

Type int

class Schema(**, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, context: dict | None = None, load_only: types.StrSequenceOrSet = (), dump_only: types.StrSequenceOrSet = (), partial: bool | types.StrSequenceOrSet = False, unknown: str | None = None*)
Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**Constraints**(***kwargs*)
Bases: *krake.data.serializable.Serializable*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *krake.data.serializable.ModelizedSchema*

class krake.data.kubernetes.**ContainerHealth**(**kwargs*)

Bases: *krake.data.serializable.Serializable*

class Schema(*, *only: types.StrSequenceOrSet | None = None*, *exclude: types.StrSequenceOrSet = ()*, *many: bool = False*, *context: dict | None = None*, *load_only: types.StrSequenceOrSet = ()*, *dump_only: types.StrSequenceOrSet = ()*, *partial: bool | types.StrSequenceOrSet = False*, *unknown: str | None = None*)

Bases: *krake.data.serializable.ModelizedSchema*

exception krake.data.kubernetes.**ObserverSchemaError**

Bases: Exception

Custom exception raised if the validation of the observer_schema fails

4.16 Client Reference

rok is a Python command line interface for the *krake* API server. It can be used to manipulate any RESTful resource handled by Krake. It can be used by end users as well as for administrative tasks.

4.16.1 Fixtures

Simple dependency injection module for rok inspired by pytest's fixtures.

There is a simple registration decorator *fixture()* that can be used to mark functions as fixtures. Functions using these fixtures can declare their dependency with the `use()` decorator. Finally, *Resolver* is used to wire fixtures and dependencies.

class rok.fixtures.**BaseUrlSession**(*base_url=None*, *raise_for_status=True*, *client_ca=None*, *ssl_cert=None*, *ssl_key=None*)

Bases: requests.sessions.Session

Simple requests session using a base URL for all requests.

Parameters

- **base_url** (*str, optional*) – Base URL that should be used as prefix for every request.
- **raise_for_status** (*bool, optional*) – Automatically raise an exception of for error response codes. Default: True

create_url(*url*)

request(*method*, *url*, **args*, *raise_for_status=None*, ***kwargs*)

Constructs a `Request`, prepares it and sends it. Returns `Response` object.

Parameters

- **method** – method for the new `Request` object.
- **url** – URL for the new `Request` object.
- **params** – (optional) Dictionary or bytes to be sent in the query string for the `Request`.

- **data** – (optional) Dictionary, list of tuples, bytes, or file-like object to send in the body of the `Request`.
- **json** – (optional) json to send in the body of the `Request`.
- **headers** – (optional) Dictionary of HTTP Headers to send with the `Request`.
- **cookies** – (optional) Dict or CookieJar object to send with the `Request`.
- **files** – (optional) Dictionary of `'filename': file-like-objects` for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*bool*) – (optional) Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol or protocol and hostname to the URL of the proxy.
- **stream** – (optional) whether to immediately download the response content. Defaults to `False`.
- **verify** – (optional) Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use. Defaults to `True`. When set to `False`, requests will accept any TLS certificate presented by the server, and will ignore hostname mismatches and/or expired certificates, which will make your application vulnerable to man-in-the-middle (MitM) attacks. Setting verify to `False` may be useful during local development or testing.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

Return type requests.Response

class rok.fixtures.**Resolver**(*fixtures=None*)

Bases: object

Dependency resolver for function arguments annotated with *depends()*.

Dependencies of a function are loaded from the `depends` attribute of the function. If a fixture is not available, the resolver checks if there is a default argument. Otherwise a `RuntimeError` is raised.

All fixtures can be overwritten by passing a corresponding keyword argument to the resolver call.

Resolver uses the context manager protocol to manage the lifecycle of generator-based fixtures.

Example

```
from sqlalchemy import create_engine
from krake.fixtures import fixture, depends, Resolver

@fixture
def engine():
    yield create_engine("postgresql://user:passwd@localhost:5432/database")

@depends("engine")
def fetch(engine, min_uid):
    with engine.begin() as connection:
        result = connection.execute(
            "SELECT username FROM users WHERE uid >= ?", min_uid
```

(continues on next page)

(continued from previous page)

```
        )
        for row in result:
            print(row["username"])

with Resolver() as resolver:
    # Execute function "fetch" with resolved fixtures. Additional
    # keyword arguments can be passed. These can also be used to
    # overwrite fixtures.
    resolver(fetch, min_uid=1000)
```

Parameters **fixtures** (*dict, optional*) – A mapping of fixture names to functions. Defaults to the mapping of `fixture.mapping`

`rok.fixtures.`**`config`**`()`

`rok.fixtures.`**`depends`**`(`**dependencies*`)`

Decorator function for marking fixture dependencies of a function.

Example

```
from rok.fixtures import fixture, depends

@depends("engine")
def fetch_records(engine):
    # Do something with the engine ...

# Fixtures themselves can also depend on other fixtures
@fixture
@depends("config")
def engine(config):
    return create_engine(config=config)

@fixture
def config:
    return load_config()
```

Parameters ***dependencies** – Fixtures the decorated function depends on

Returns Decorator for explicitly marking function dependencies.

Return type callable

`rok.fixtures.`**`fixture`**`(`*func*`)`

Mark a function or generator as fixtures. The name of the function is used as fixture name.

If the marked function is a generator function, the fixture can be used as kind of context manager:

```
@fixture
def session():
    with Session() as session:
        yield session
```

`rok.fixtures.`**`mapping`**

Mapping of registered fixture names to functions

Type dict

Parameters **func** – Function that should be registered as fixture

Raises RuntimeError – If the a fixtures with the same name is already registered.

rok.fixtures.**session**(*config*)

4.16.2 Command Line Parser

This module defines a declarative API for Python's standard argparse module.

class rok.parser.**MetricAction**(**args*, *nargs=None*, *default=None*, *metavar=None*, ***kwargs*)

Bases: argparse.Action

argparse action for metric values

A metric argument requires two arguments. The first argument is the name of a metric (str). The second argument is the weight of the argument as float. The option can be called several times.

Example

```
cli --metric-argument my-metric 1.2 --metric-argument my-other-metric 4.5
```

The action will populate the namespace with a list of dictionaries:

```
[
    {"name": "my-metric", "weight": 1.2},
    {"name": "my-other-metric", "weight": 4.5},
    ...
]
```

class rok.parser.**ParserSpec**(**args*, ***kwargs*)

Bases: object

Declarative parser specification for Python's standard argparse module.

Example

```
from rok.parser import ParserSpec, argument

spec = ParserSpec(prog="spam", description="Spam command line interface")

@spec.command("spam", help="Spam your shell")
@argument("-n", type=int, default=42, help="How often should I spam?")
@argument("message", help="Spam message")
def spam(n, message):
    for _ in range(n):
        print(message)

parser = spec.create_parser()
args = parser.parse_args()
```

Specifications can be nested:

```
eggs = ParserSpec("eggs", aliases=["eg"], help="... and eggs")

@eggs.command("spam")
def eggs_spam():
    while True:
        print("spam")
        print("eggs")

spec.add_spec(eggs)
```

Parameters

- ***args** – Positional arguments that will be passed to either `argparse.ArgumentParser` or subparsers.
- ***kwargs** – Keyword arguments that will be passed to either `argparse.ArgumentParser` or subparsers.

add_spec(*subparser*)
Register another specification as subparser

Parameters **subparser** (`ParserSpec`) – Sub-specification defining subcommands

command(*name*, **args*, ***kwargs*)
Decorator function for commands registering the name, positional and keyword arguments for a subparser.

Parameters

- **name** (*name*) – Name of the command that will be used in the command line.
- ***args** – Positional arguments for the subparser
- ****kwargs** – Keyword arguments for the subparser

Returns Decorator for functions that will be registered as `command` default argument on the subparser.

Return type callable

create_parser(*parent=None*)
Create a standard Python parser from the specification

Parameters **parent** (*optional*) – argparse subparser that should be used instead of creating a new root `argparse.ArgumentParser`

Returns Standard Python parser

Return type argparse.ArgumentParser

subparser(**args*, ***kwargs*)
Create a subspecification and automatically register it via *add_spec()*

Parameters

- ***args** – Positional arguments for the specification
- ****kwargs** – Keyword arguments for the specification

Returns The new subspecification for subcommands

Return type *ParserSpec*

class rok.parser.**StoreDict**(*option_strings*, *dest*, *nargs=None*, *metavar='KEY=VALUE'*, ***kwargs*)

Bases: argparse.Action

Action storing <key=value> pairs in a dictionary.

Example

```
parser = argparse.ArgumentParser()
parser.add_argument(
    '--foo', action=StoreDict
)
args = parser.parse_args('--foo label=test --foo lorem=ipsum')
assert argparse.Namespace(foo={'label': 'test', 'lorem': 'ipsum'}) == args
```

rok.parser.**arg_backoff**(*fn*)

rok.parser.**arg_backoff_delay**(*fn*)

rok.parser.**arg_backoff_limit**(*fn*)

rok.parser.**arg_formatting**(*fn*)

rok.parser.**arg_global_metric**(*fn*)

rok.parser.**arg_labels**(*fn*)

rok.parser.**arg_metric**(*fn*)

rok.parser.**arg_namespace**(*fn*)

rok.parser.**argument**(**args*, ***kwargs*)

Decorator function for standard argparse arguments.

The passed arguments and keyword arguments are stored as tuple in a parser_arguments attribute of the decorated function. This list will be reused by class:*ParserSpec* to add arguments to decorated commands.

Parameters

- ***args** – Positional arguments that should be passed to argparse.ArgumentParser.add_argument().
- ****kwargs** – Keyword arguments that should be passed to argparse.ArgumentParser.add_argument().

Returns A decorator that can be used to decorate a command function.

Return type callable

rok.parser.**mutually_exclusive_group**(*group*)

Decorator function for mutually exclusive argparse arguments.

Parameters **group**(*list of tuples*) – A list of the standard :mod: *argparse* arguments which are mutually exclusive. Each argument is represented as a tuple of its args and kwargs.

Returns A decorator that can be used to decorate a command function.

Return type callable

Python Module Index

k

r

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