Redirectory

Release 1.0.0

Jul 03, 2019

Contents

Redirectory is a tool that manages redirects on a cluster level. Requests that would usually end in a **404 PAGE NOT FOUND** can now redirect to new pages specified with custom rules. It binds itself as the default backend (essential a wild card) of your ingress controller and catches all the request that the cluster can't find an ingress rule for.

KEY FEATURES

1. Build to run in Kubernetes.
2. Easily scalable by spawning new workers.
3. Can handle multiple domains and sub-domains in a cluster.
4. Every redirect is represented by a redirect rule. Redirect rules support regex.
5. Regex matching performed by Intel's open source Hyperscan regex engine.
6. Can construct new urls by extracting part of old url. For example get an id from the old url and place it in the new one.
7. UI - Easy to use interface so that your marketing people can use it as well.

AUTHOR **Kumina B.V.** (Ivaylo Korakov)

CHAPTER 1

Install

Install Redirectory and creates all the needed resources for it from scratch.

```
helm install --name=redirectory redirectory/conf/helm
```

For more info on installation take a look at the *Installation*.

1.1 Documentation

This part of the documentation will show you how to get started using Redirectory.

1.1.1 Overview

The problem

A lot of big companies have large websites that are constantly changing and are dynamic. This is really nice in order to keep you brand/site up to date with new trends but it also has a bad side effect. Old web pages get deleted and people opening them are getting 404 errors. Usually companies are familiar with that and they even know which old url should redirect to which new one but unfortunately there isn't an easy way to do that in kubernetes at the moment.

The solution

The **Redirectory for Kubernetes** project aims to solve this problem once and for all of the companies. It aims to provide a set of features which makes it easy for people of Kumina or customers of Kumina to manage their redirects on their Kubernetes clusters. The project will live on the ingress level in a cluster and will intercept all requests that the ingress is not able to serve and otherwise would send out a 404. Redirectory will catch those errors and try to find the best new url to redirect to in order for the customer to have a seamless experience even though they might be using old and inactive urls.

1.1.2 Usage

This part of the documentation assumes you already have Redirectory setup and running on a Kubernetes cluster and you have access to the User Interface provided by the management pod.

Overview

This is a piece of software for redirecting requests that would usually end up with a 404 response to a new destination specified by given rules. It is made to work and take advantage of a Kubernetes environment. What you are currently looking at is the so called "management panel" or whatever you would like to call it.

From here you can manage amd access all of the features provided by Redirectory. This User Guide aims to show you how you can use it! Lets begin with the rules.

Rules

Rules are the main things that tells Redirectory how to redirect the incoming requests. This section will show you how to:

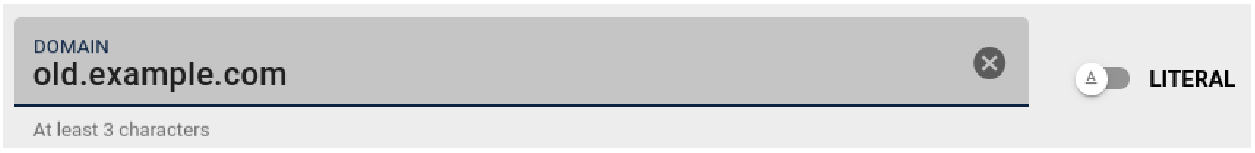
1. Create new rules
2. Exit existing rules
3. And delete not needed once

In order for it to redirect lets say:

```
https://old.example.com/.* -> to -> https://new.example.com/
```

we will first need to enter a rule for this. First you will have to go to the Redirect Rule Explorer section.

There underneath the search filters you will find a button **CREATE NEW REDIRECT RULE**: Once clicked a menu with a few options will appear. The first thing to specify is the domain you would like to redirect from. Keep in mind this domain should be configured that it points to the cluster you are using Redirectory in. After you are done with the domain it should look something like this:


DOMAIN
old.example.com
At least 3 characters

The next thing we need to configure is the path of the domain we just added. Lets to this one the same way as the domain. You might have noticed that we have a **(.*)** in the path of the rule.

This is called **Regex** and it is one of the features of Redirectory, If you have a regex expression you need to toggle to switch between **Regex** and **Literal**

See a little bit more info on Regex in the note below.

Note: **REGEX** A really simple tutorial.

Regex is quite an expansive topic we don't need much to be able to use it. It is used to select text and in our case URLs. Here are most of the things you will need to get started:

syntax	meaning
.	any character
\d	just numbers
\w	letters and numbers
*	zero or more
+	one or more

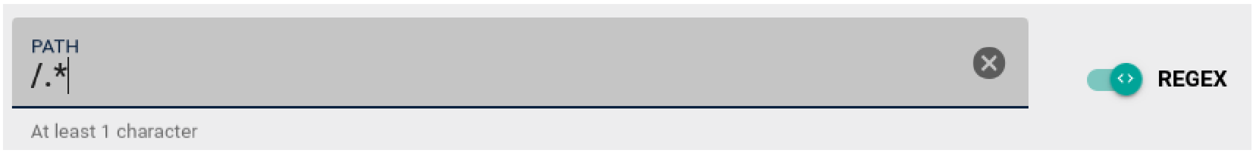
Now we can chain them together like this:

```
/test/path.*
```

which will match any of those:

```
/test/path/any
/test/path/of
/test/path/those
/test/path/123
```

Now that we now what we are actually typing in we can fill it in and it should look like the following:



You can fill in the destination the exact same way we did the first two. The last thing that needs to be configured is the weight of a rule. Why do we need it? Sometimes you can get conflicting rules that both of them match the same request. When this happens Redirectory has to know which rules has bigger weight (priority). This is expressed with the weight value of the rule. By default all rules get a weight of 100.

Now we can just create the rule with the **CREATE** button.

Redirect Rule Explorer

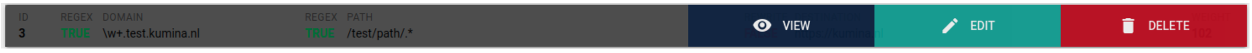
With the Explorer you have all the things you would need in order to manage all of the Redirect Rules for Redirectory. Like we discussed in the Rules section here you can create a new rule but also much more.

On top are the filters. With them you can search through all of the rules you have. You can stack multiple filters to narrow down your search even more. Also keep in mind that for the domain, path and destination filters you can use (*) which is an fnmatch.

Note: FNMATCH or also called Function Match is a way simpler form of regex. Basically you can have a **(*)** which is equivalent to **(.+)** in Regex and and will match one or more.

After you set the filters just press the button **APPLY FILTERS**.

Once you have located the rule that you want in order to view it, edit or delete it you can just click on it: Then the following options will be given for that rule:



ID	REGEX	DOMAIN	REGEX	PATH
3	TRUE	\w+.test.kumina.nl	TRUE	/test/path/.*

Keep in mind the rules are not updated automatically in the User Interface. To make sure your are seeing the latest changes to the rules please click the **REFRESH PAGES** button.

Bulk Import

But what if I have a lot of rules? For this situation you can make use of the bulk import feature. With it you can upload a CSV (Coma Separated Values) file and all of the rules will be added at once. Because CSV is a basic format a lot of programs support an export to it. You will have to refer to the documentation of the program you are using for more information on exporting the data as CSV.

Take a look at the Bulk Import Section for more information on how the CSV file should be formated in order to get the smooth import.

Once you have uploaded the file the import will begin immediately. The time it takes to process and add all the rules varies on how of course how many you have.

Ambiguous requests

Ambiguous requests are requests for which Redirectory was unable to decide 100% of what should be the final destination. What does this mean? The main reason of you seeing ambiguous requests is that you have some rules that are not configured correctly.

Sometimes it happens that two or more rules intersect each other and Regex has trouble choosing which one is the more important one because all of them match. Example of intersection:

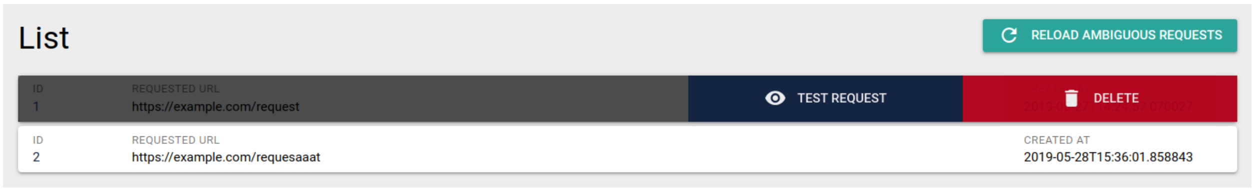
```
1. ggg.test.kumina.nl/test/path/.*
2. \\w+.test.kumina.nl/test/path/.*
3. .*.test.kumina.nl/test/pa.*
```

Now if we make a requests that looks like this:

```
ggg.test.kumina.nl/test/path/aaabb
```

we will match all of the three rules and Redirectory will not know which one should it choose. When this happens Redirectory will always choose the first rule (with the smallest id) and it will also save the request as ambiguous in order for a person to take a look and change the weights of the rules in order not to happen again.

You will be able to see the ambiguous requests section. There are a few options you can make use of in this section. On the top right there is the **RELOAD AMBIGUOUS REQUESTS** button: Once you click an entry/request you are presented with two options. Test option will put this request in the Test Section and show you what is happening behind the scenes. From there you can specify the correct weights for the rules in order to avoid any ambiguous requests in the future. Once you have fixed the issue for a given ambiguous request you can delete it with the second option. See image below for better understanding.



ID	REQUESTED URL	CREATED AT
1	https://example.com/request	
2	https://example.com/requesaaat	2019-05-28T15:36:01.858843

Hyperscan Database

You have probably noticed that when adding, updating and deleting a rule you have a message that say that the changes will not apply until you compile a new Hyperscan database. This is due to the backend and how Hyperscan works. First make all the changes you would like and then once you are done with all of them you can compile/create a new Hyperscan database.

The settings are located in the Hyperscan Database and Workers Section. Now that you have made the changes you wanted to the rules you can press the **COMPILE NEW HS DB** button. This will create a new Hyperscan Database and apply it to all of the workers. That is everything you need to be worried about with Hyperscan. If you are interested in the workers and how they work please take a look at the next section in the User Guide.

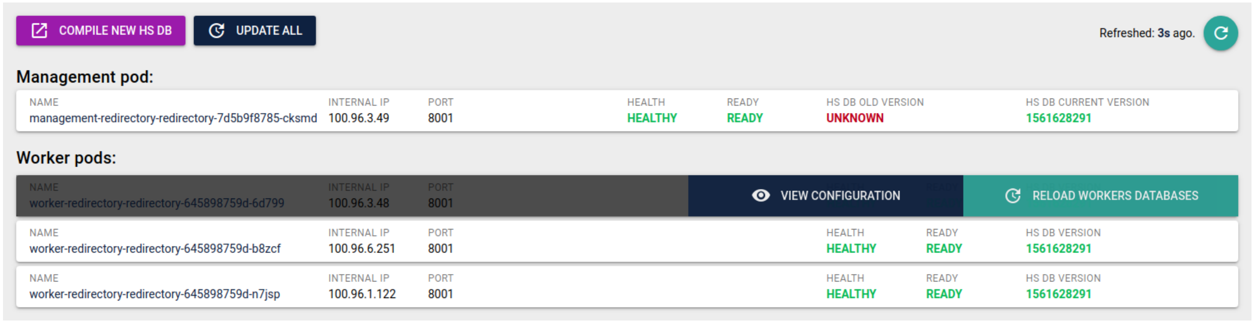
Workers and Kubernetes

Redirectory is an application that runs in Kubernetes and makes use of it's scaling features. That is why the application is split into two parts: **management** and **workers**.

The workers are the one that process all of the incoming requests. That is why they need to be up to date with the newest version of the Hyperscan Database. In other words the Redirect Rules.

You will find all of the options for the management and workers in the Hyperscan Database and Workers Section. From there you can see the status of each worker and the current database they have loaded on them. This information updates automatically every 10 seconds or you can click the **REFRESH** button to update now.

The **COMPILE NEW HS DB** button creates a new database and **updates all the workers** after that. If for some reason a worker is out of date you can use the **UPDATE ALL** button or by clicking on the out of date workers and updating it individually. From there you can view the configuration of the workers as well. Take a look at the picture below:



Management pod:

NAME	INTERNAL IP	PORT	HEALTH	READY	HS DB OLD VERSION	HS DB CURRENT VERSION
management-redirectory-redirectory-7d5b9f8785-cksmd	100.96.3.49	8001	HEALTHY	READY	UNKNOWN	1561628291

Worker pods:

NAME	INTERNAL IP	PORT	HEALTH	READY	HS DB VERSION
worker-redirectory-redirectory-645898759d-6d799	100.96.3.48	8001			
worker-redirectory-redirectory-645898759d-b8zcf	100.96.6.251	8001	HEALTHY	READY	1561628291
worker-redirectory-redirectory-645898759d-n7jsp	100.96.1.122	8001	HEALTHY	READY	1561628291

1.1.3 Installation

The application is made to run on a **Kubernetes** cluster. There are a few things you need to have in order to deploy it.

1. **Persistent Volume** - In order for the management pod to store the rules (data) in case of a failure or restart. Workers don't have persistent volumes. They sync their data from the management pod.
2. **Role bindings** - Needed because the application must know of worker and management pods. The following permissions are needed for a Role resource:

resources	**verbs**
endpoints	get, list, watch
pods	get, list, watch

You would be able to find all the **.yaml** configuration files in the Redirectory repository.

Installation manually

This installation method is NOT recommended! All of the needed configuration files are located under the folder:

```
$ redirectory/conf/kubernetes
```

You will have to apply all the files manually to your cluster with the following command:

```
$ kubectl apply -f management_ingress.yaml
$ kubectl apply -f management_svc.yaml -f worker_svc.yaml
... and so on
```

You may or may not need to edit the configuration files to fit your particular setup.

Installation with HELM

To make the installation easier we are making use of HELM. It is a soft of package manager for Kubernetes but more like a templating engine for Kubernetes **.yaml** configuration files.

If you are not familiar with HELM please take a look at theirs documentation on how to use it: HELM docs

Before continuing make sure you have HELM installed on your kubernetes cluster.

Install

Install Redirectory and creates all the needed resources for it from scratch.

```
$ helm install --name=redirectory redirectory/conf/helm
```

Update

Updates only the resources/things that have changes since the last update or install of Redirectory

```
$ helm upgrade redirectory redirectory/conf/helm
```

Delete

Deletes Redirectory from the Kubernetes cluster.

Warning: When deleting the application like this it will also DELETE ALL it's data. You will not be able to get the data back.

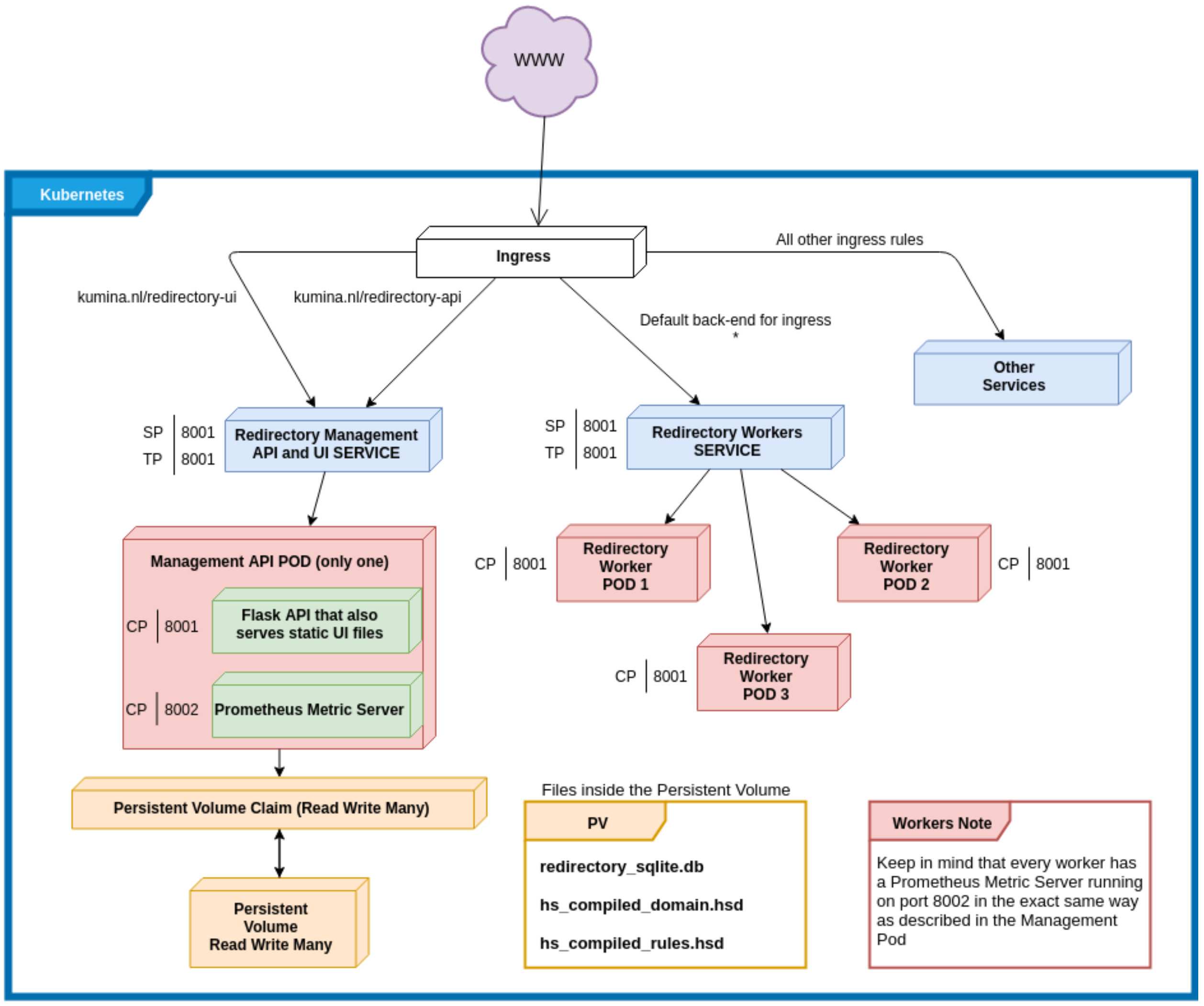
```
$ helm delete --purge redirectory
```

1.1.4 Kubernetes

Redirectory is meant to run in a Kubernetes cluster. Kubernetes is a really huge topic and it will not be covered in this documentation. Let's call it a pre-requisite. If you would like to get started you can check the official get started guide.

The application is split into two parts. The worker pods which only handle redirecting requests and a management pod which handles all other functionalities of the application.

To gain a better understanding of how the application runs in Kubernetes please refer to the diagram below.


WWW
Kubernetes
Ingress
All other ingress rules
kumina.nl/redirectory-ui
kumina.nl/redirectory-api
Default back-end for ingress
*
Other Services
SP 8001
TP 8001
Redirectory Management API and UI SERVICE
SP 8001
TP 8001
Redirectory Workers SERVICE
Management API POD (only one)
CP 8001
Flask API that also serves static UI files
CP 8002
Prometheus Metric Server
CP 8001
Redirectory Worker POD 1
Redirectory Worker POD 2
CP 8001
CP 8001
Redirectory Worker POD 3
Persistent Volume Claim (Read Write Many)
Persistent Volume Read Write Many
Files inside the Persistent Volume
PV
redirectory_sqlite.db
hs_compiled_domain.hsd
hs_compiled_rules.hsd
Workers Note
Keep in mind that every worker has a Prometheus Metric Server running on port 8002 in the exact same way as described in the Management Pod

1.1.5 Testing

For the Redirectory project unit testing is encouraged! The library of choice to help us with implementing the unit tests is called **PyTest** and you can see their docs at: pytest docs.

Set up

Before we start testing Redirectory let's setup our testing environment. There is already a nice `requirements_test.txt` file we can use for this. You can create an environment with the following moment:

```
mkvirtualenv redirectory_test -r requirements_test.txt
```

Running the tests

We can run the tests with the following command:

```
PYTHONPATH=. pytest
```

and if you would like to see the `stdout` while the tests are running:

```
PYTHONPATH=. pytest -s
```

Structure

Because we make use of **pytest** the tests folder is split into two as shown bellow:

```
tests
├── cases
│   ├── database
│   └── hyperscan
├── fixtures
│   ├── configuration.py
│   ├── database_ambiguous.py
│   ├── database_empty.py
│   ├── database_populated.py
│   └── hyperscan.py
```

Fixtures are functions that will run before every test. Let's say that a certain test needs an already loaded empty database in order to run. We can create a fixture `database_empty` and add it as a requirement to this particular unit test.

This is how the `database_empty` fixture would look like:

```
@pytest.fixture
def database_empty(configuration):
    # Import DB Manager first before the models
    from redirectory.libs_int.database import DatabaseManager

    # Import the models now so that the DB Manager know about them
    import redirectory.models

    # Delete any previous creations of the Database Manager and tables
    DatabaseManager().reload()
    DatabaseManager().delete_db_tables()

    # Create all tables based on the just imported modules
    DatabaseManager().create_db_tables()
```

Tip: Fixtures can be added as requirements for other fixtures. In this case before we can init the database we need to make sure the `configuration` is available.

and the unit test will look like this:

```
def test_add_ambiguous_request(self, database_empty):
    """
    Test Description ...
    """
    # Get session
    from redirectory.libs_int.database import DatabaseManager
    db_session = DatabaseManager().get_session()
```

(continues on next page)

(continued from previous page)

```
    # Here is your actual test
    assert True

    # Return session
    DatabaseManager().return_session(db_session)
```

Must do

Always return the session to the database before your the end of your test

1.1.6 License

Redirectory is released under the BSD 3 Clause.

1.2 API Reference

If you are interested in information about a class, specific function or more this is the place to take a look.

1.2.1 Redirectory API Reference

This part of the documentation is for developers who would like to know the insides of the project. Here you will find all of the documentation of the source code of Redirectory.

The project is split into different packages for better structure.
Here is a quick overview of all of the packages that the project consists of:

libs_int overview

Libs_int is the main package that holds most of the main logic of the application. The main goal is to move out the logic from the API endpoints themselves and have it in one place. This package holds logic for quite a few things:

1. **Config** - .yaml configuration files
2. **Database** - all the needed classes and methods to interact with the database
3. **Hyperscan** - all of the logic of the Hyperscan regex engine
4. **Importers** - different file importers. At the moment only CSV.
5. **Metrics** - logic about Prometheus metrics
6. **Service** - helper classes and methods for API functionality. Also Gunicorn.

models overview

Redirectory uses a **SQLite3** database which sits as a file in the **data folder** of the application. The Models packages contains the different models for the database. Redirectory is using **SQLAlchemy** library to the it's interactions with the database.

runnables overview

Again because Redirectory is made for Kubernetes we split up the application in three different parts:

1. Management
2. Worker
3. Compiler

Because of this we need a nice way to separate between those different modes. Here the **runnables** come in play. A runnable is a class which makes use of the `run()` method which loads different things and prepares the application to run in the correct mode.

services overview

The service package is where all of the different API endpoints are situated. Because the application is made for Kubernetes there are a few different modes that Redirectory can run as. Therefore the API endpoints are split in the same manner:

1. Management Endpoints
2. Worker Endpoints

Based on the **node_type** which is specified in the **config.yaml** the different sets of API endpoints are loaded at startup. In other words, if you run the application as **management** you won't be able to call **worker** endpoints and the other way around.

Important: Keep in mind the **stats** endpoints are loaded in both **management and worker** mode.

Contents

Libs_Int package

redirectory.libs_int.config package

redirectory.libs_int.config.configuration module

class redirectory.libs_int.config.configuration.**Configuration**

Bases: `object`

path = None

values = None

redirectory.libs_int.database package

redirectory.libs_int.database.database_actions module

class redirectory.libs_int.database.database_actions.**NoneType**

Bases: `object`

redirectory.libs_int.database.database_actions.**encode_model**(*model: sqlalchemy.ext.declarative.api.DeclarativeMe*, *parent_class: Any = None*, *expand: bool = False*) → dict

Encodes a DB instance object of a given model into json

Parameters

- **model** – The DB model instance to serialize to json
- **parent_class** – A DB model might inherit from another DB model. Pass the parent class in order to be serialized correctly
- **expand** – to include relationships or not

Returns a dictionary with basic data types that are all serializable

redirectory.libs_int.database.database_actions.**encode_query**(*query: list*, *expand: bool = False*) → list

Loops through all of the objects in a query and encodes every object with the help of encode_model() function. All of the individual encoded models are added into a list and then returned.

Parameters

- **query** – the query that you would like to encode
- **expand** – if you should expand relationships in the models

Returns a list of dictionaries which are the encoded objects

redirectory.libs_int.database.database_actions.**get_or_create**(*session*, *model*, *defaults=None*, ***kwargs*)

Gets an instance of an object or if it does not exist then create it.

Parameters

- **session** – the database session
- **model** – the model / table to ger or create from
- **defaults** – any default parameters for creating
- ****kwargs** – the criteria to get or create

Returns a tuple(p,q) p: an instance of the object and q: if it is new or old

redirectory.libs_int.database.database_actions.**get_table_row_count**(*db_session*, *model_table*) → int

Gets the number of rows in a given database in the given database session

Parameters

- **db_session** – the database session to use for db actions
- **model_table** – the model / table

Returns integer represent the number of row in the table

redirectory.libs_int.database.database_actions.**sanitize_like_query**(*query_str: str*) → str

Sanitizes a string to be used as a query in the DB. It will replace a * with % only when the star is not escaped.

Parameters **query_str** – original string to sanitize

Returns sanitized converted string

redirectory.libs_int.database.database_manager module

class redirectory.libs_int.database.database_manager.**DatabaseManager**

Bases: object

create_db_tables()

Will create all model's tables associated with the current DatabaseManager base. The creation of those tables is safe. If a table already exists it will not be created again. If the DatabaseManager is not initialized then a ValueError will be raised.

delete_db_tables()

Will drop all tables associated with the current base of the DatabaseManager. Every model's table that inherits from this base will be dropped. If the DatabaseManager is not initialized then a ValueError will be raised.

get_base()

Gets the current base that all models should inherit from. Once a model inherits from this base it will be associated with it.

Returns the current base

get_session()

Creates a scoped session with with the help of the session maker. This session is specific to the current thread from where this function is called. If a session already exists it will be returned but if not a new one will be created. If the DatabaseManager is not initialized then a ValueError will be raised.

Returns a database session for the current thread

reload()

return_session(*session*)

Closes the given session and removes it from DatabaseManager to prevent from any further use. Sets the session of the DatabaseManager to None

Parameters **session** – the session to remove

`redirectory.libs_int.database.database_manager.`**`get_connection_string`**`()`

Generates a connection string to be passed to SQLAlchemy. The string is created from the current loaded configuration with the help of the Configuration() class. There are two options for both SQLite and MySQL database connections.

Returns a connection string for SQLAlchemy to use for an engine

redirectory.libs_int.database.database_pagination module

class `redirectory.libs_int.database.database_pagination.`**`Page`**`(`*items*, *page*, *page_size*, *total*`)`

Bases: `object`

`redirectory.libs_int.database.database_pagination.`**`paginate`**`(`*query*, *page: int*, *page_size: int*`)` → redirectory.libs_int.database.database_pagination.Page

Creates a query with the help of limit() and offset() to represent a page. Also counts the total number of items in the given database.

Parameters

- **query** – the query which specifies the model tha paginate
- **page** (*int*) – the page number
- **page_size** (*int*) – how many items per page

Returns a Page object with all the items inside

redirectory.libs_int.database.database_rule_actions module

redirectory.libs_int.hyperscan package

redirectory.libs_int.hyperscan.hs_actions module

redirectory.libs_int.hyperscan.hs_database module

redirectory.libs_int.hyperscan.hs_manager module

redirectory.libs_int.hyperscan.search_context module

class `redirectory.libs_int.hyperscan.search_context.`**`SearchContext`**`(`*original: str*, ***kwargs*`)`

Bases: `dict`

`handle_match``(`*destination_id: int*, *from_index: int*, *to_index: int*`)`

Handles a hyperscan matched passed from the match_event_handler. If the length of the match matches the length of the original search query it will be added to the matched_ids.

Parameters

- **destination_id** – the id of the matched expression from Hyperscan

- **`from_index`** – from where the match starts
- **`to_index`** – until where the match ends

`is_empty`()

Checks in any matches have been found associated with this context

Returns a boolean representing if any matches are found

`matched_ids = None`

`original = None`

redirectory.libs_int.importers package

This package/folder contains the different importers Redirectory has.

At the moment only one is available but more may be added in the future if they are requested or people contribute.

Follow the links bellow to see the API of the importer.

redirectory.libs_int.importers.csv_importer module

redirectory.libs_int.metrics package

redirectory.libs_int.metrics.metrics module

`redirectory.libs_int.metrics.metrics.`**`start_metrics_server`**()

Starts a http server on a port specified in the configuration file and exposes Prometheus metrics on it. Also removes GC_COLLECTOR metrics because they are not really needed.

redirectory.libs_int.service package

redirectory.libs_int.service.api module

`class` `redirectory.libs_int.service.api.`**`Api`**(*app=None, version='1.0', title=None, description=None, terms_url=None, license=None, license_url=None, contact=None, contact_url=None, contact_email=None, authorizations=None, security=None, doc='/', default_id=<function default_id>, default='default', default_label='Default namespace', validate=None, tags=None, prefix='', ordered=False, default_mediatype='application/json', decorators=None, catch_all_404s=False, serve_challenge_on_401=False, format_checker=None, **kwargs*)

Bases: `flask_restplus.api.Api`

`base_path`

The API path

Return type str

redirectory.libs_int.service.api_actions module

`redirectory.libs_int.service.api_actions.`**`api_error`**(*message: str, errors: Union[str, list], status_code: int*)

Returns an api error with a given status and a message/messages

Parameters

- **`message`** – A overall message of the error. E.g. Wrong input.
- **`errors`** – A message in str format or a list of strings for multiple error messages
- **`status_code`** – The status of the error. E.g. 404, 503 ..

redirectory.libs_int.service.gunicorn_server module

`class` `redirectory.libs_int.service.gunicorn_server.`**`GunicornServer`**(*app, options=None*)

Bases: `gunicorn.app.base.BaseApplication`

This class provides the ability to run gunicorn server from inside of python instead of the running it through the command prompt Gives you a nicer way to handle it and you can override key methods to make it more specific for our use case

`static get_number_of_workers`(*is_worker: bool = False*)

Calculates the number of workers the gunicorn server will use

`init`(*parser, opts, args*)

`load`()

`load_config`()

This method is used to load the configuration from one or several input(s). Custom Command line, configuration file. You have to override this method in your class.

`static load_metric_server`()

When run() is called on Gunicorn it starts a new process with this flask App. The metric server must run in the same process as the Flask API in order to share the metrics. This function starts the metric server when the Flask APP is loaded into the new process.

redirectory.libs_int.service.namespace_manager module

`class` `redirectory.libs_int.service.namespace_manager.`**`NamespaceManager`**

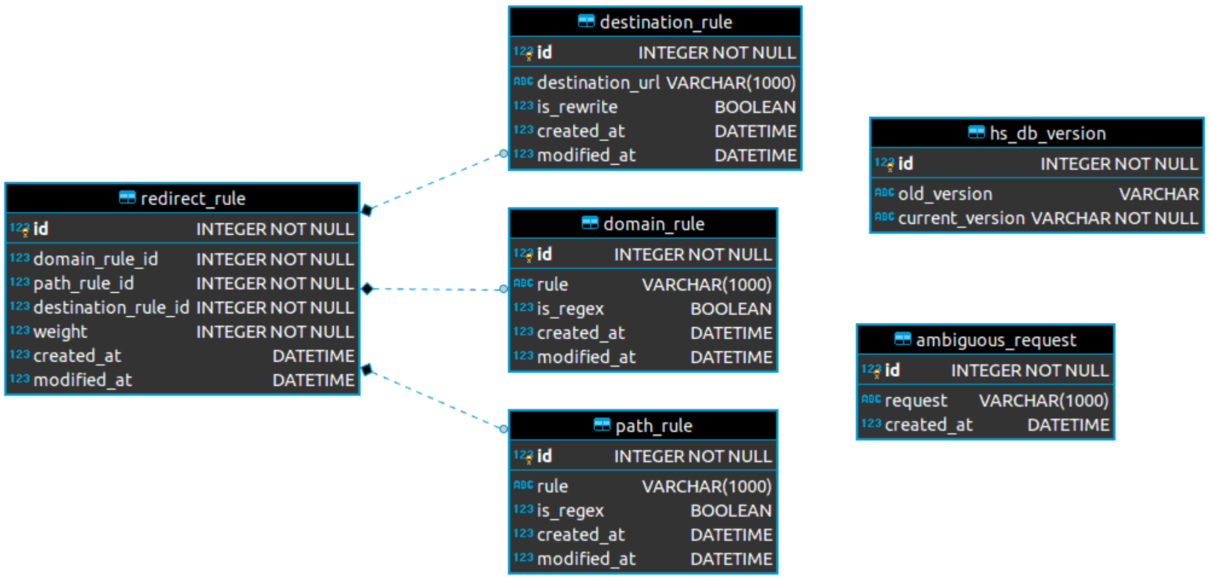
Bases: `object`

`get_namespace`(*name: str*)

`namespace_map = {}`

Models package

Here as a diagram of the database followed by their corresponding classes. I think they are simple enough to understand directly ;)



Redirect Rule

```python
from datetime import datetime

from kubi_ecs_logger import Logger, Severity
from sqlalchemy import Column, Integer, DateTime, ForeignKey, UniqueConstraint
from sqlalchemy.orm import relationship

from redirectory.libs_int.database import DatabaseManager

base = DatabaseManager().get_base()


class RedirectRule(base):
    __tablename__ = "redirect_rule"

    id = Column(Integer, autoincrement=True, primary_key=True)

    domain_rule_id = Column(Integer, ForeignKey('domain_rule.id'), nullable=False)
    domain_rule = relationship("DomainRule", lazy="joined", foreign_keys=[domain_rule_id])

    path_rule_id = Column(Integer, ForeignKey("path_rule.id"), nullable=False)
    path_rule = relationship("PathRule", lazy="joined", foreign_keys=[path_rule_id])

    destination_rule_id = Column(Integer, ForeignKey("destination_rule.id"), nullable=False)
    destination_rule = relationship("DestinationRule", lazy="joined", foreign_keys=[destination_rule_id])

    weight = Column(Integer, nullable=False, default=100)

    created_at = Column(DateTime, default=datetime.now())
    modified_at = Column(DateTime, default=datetime.now())
```

(continues on next page)

(continued from previous page)

```

    __table_args__ = (UniqueConstraint('domain_rule_id', 'path_rule_id', 'destination_
↪rule_id',
                                       name='_domain_path_destination_uc'),)

    def modify(self):
        self.modified_at = datetime.now()

    def delete(self, db_session, safe: bool = True):
        db_session.delete(self)
        db_session.commit()

        Logger() \
            .event(category="database", action="redirect rule deleted") \
            .log(original=f"Redirect rule with id: {self.id} has been deleted") \
            .out(severity=Severity.DEBUG)

        self.domain_rule.delete(db_session, safe=safe)
        self.path_rule.delete(db_session, safe=safe)
        self.destination_rule.delete(db_session, safe=safe)
```

Path Rule

```python
from datetime import datetime

from kubi_ecs_logger import Logger, Severity
from sqlalchemy import Column, Integer, DateTime, String, Boolean, UniqueConstraint,␣
↪select, func

from redirectory.libs_int.database import DatabaseManager

base = DatabaseManager().get_base()


class PathRule(base):
    __tablename__ = "path_rule"

    id = Column(Integer, autoincrement=True, primary_key=True)
    rule = Column(String(1000))
    is_regex = Column(Boolean, default=False)
    created_at = Column(DateTime, default=datetime.now())
    modified_at = Column(DateTime, default=datetime.now())
    __table_args__ = (UniqueConstraint("rule", "is_regex", name="_rule_regex_uc"),)

    def modify(self):
        self.modified_at = datetime.now()

    def delete(self, db_session, safe: bool = True):
        if safe:
            from redirectory.libs_int.database import get_usage_count
            if get_usage_count(db_session, type(self), self.id) > 0:
                return

        db_session.delete(self)
```

(continues on next page)

(continued from previous page)

```
        db_session.commit()

        Logger() \
            .event(category="database", action="path deleted") \
            .log(original=f"Path with id: {self.id} has been deleted") \
            .out(severity=Severity.DEBUG)
```

Domain Rule

```
from datetime import datetime

from kubi_ecs_logger import Logger, Severity
from sqlalchemy import Column, Integer, DateTime, String, Boolean, UniqueConstraint,␣
↪select, func

from redirectory.libs_int.database import DatabaseManager

base = DatabaseManager().get_base()


class DomainRule(base):
    __tablename__ = "domain_rule"

    id = Column(Integer, autoincrement=True, primary_key=True)
    rule = Column(String(1000))
    is_regex = Column(Boolean, default=False)
    created_at = Column(DateTime, default=datetime.now())
    modified_at = Column(DateTime, default=datetime.now())
    __table_args__ = (UniqueConstraint("rule", "is_regex", name="_rule_regex_uc"),)

    def modify(self):
        self.modified_at = datetime.now()

    def delete(self, db_session, safe: bool = True):
        if safe:
            from redirectory.libs_int.database import get_usage_count
            if get_usage_count(db_session, type(self), self.id) > 0:
                return

        db_session.delete(self)
        db_session.commit()

        Logger() \
            .event(category="database", action="domain deleted") \
            .log(original=f"Domain with id: {self.id} has been deleted") \
            .out(severity=Severity.DEBUG)
```

Destination Rule

```
from datetime import datetime

from kubi_ecs_logger import Logger, Severity
```

(continues on next page)

(continued from previous page)

```
from sqlalchemy import Column, Integer, DateTime, String, Boolean, UniqueConstraint,␣
↪select, func

from redirectory.libs_int.database import DatabaseManager

base = DatabaseManager().get_base()


class DestinationRule(base):
    __tablename__ = "destination_rule"

    id = Column(Integer, autoincrement=True, primary_key=True)
    destination_url = Column(String(1000))
    is_rewrite = Column(Boolean, default=False)
    created_at = Column(DateTime, default=datetime.now())
    modified_at = Column(DateTime, default=datetime.now())
    __table_args__ = (UniqueConstraint("destination_url", "is_rewrite", name="_
↪destination_rewrite_uc"),)

    def modify(self):
        self.modified_at = datetime.now()

    def delete(self, db_session, safe: bool = True):
        if safe:
            from redirectory.libs_int.database import get_usage_count
            if get_usage_count(db_session, type(self), self.id) > 0:
                return

        db_session.delete(self)
        db_session.commit()

        Logger() \
            .event(category="database", action="destination deleted") \
            .log(original=f"Destination with id: {self.id} has been deleted") \
            .out(severity=Severity.DEBUG)
```

Ambiguous Requests

```
from datetime import datetime

from sqlalchemy import Column, Integer, DateTime, String

from redirectory.libs_int.database import DatabaseManager

base = DatabaseManager().get_base()


class AmbiguousRequest(base):
    __tablename__ = "ambiguous_request"

    id = Column(Integer, autoincrement=True, primary_key=True)
    request = Column(String(1000), unique=True)
    created_at = Column(DateTime, default=datetime.now())
```

Hyperscan DB Version

```python
from sqlalchemy import Column, Integer, String

from redirectory.libs_int.database import DatabaseManager

base = DatabaseManager().get_base()


class HsDbVersion(base):
    __tablename__ = "hs_db_version"

    id = Column(Integer, autoincrement=True, primary_key=True)
    old_version = Column(String, nullable=True)
    current_version = Column(String, nullable=False)
```

Runnables package

redirectory.runnables.compiler module

redirectory.runnables.management module

redirectory.runnables.runnable module

class redirectory.runnables.runnable.**Runnable**

Bases: abc.ABC

config = None

run()

redirectory.runnables.runnable_service module

redirectory.runnables.worker module

Services package

This package contains all endpoints that Redirectory has. Just like other parts of the application the API Endpoints are also split into different parts:

1. **Management** - All endpoints for management and UI
2. **Worker** - All endpoints for workers
3. **Status** - Endpoints for watching the status of the application
4. **Root** - Endpoints that are bound to **/** (root path). UI for **management** and redirect for **worker**

Contents

Worker Endpoints

Worker Get HS DB Version Endpoint

Worker Reload HS DB Endpoint

Status Endpoints

Status Health Check Endpoint

Status Readiness Check Endpoint

Status Get Node Configuration Endpoint

Management Ambiguous Endpoints

Management Add Ambiguous Request Endpoint

Management Delete Ambiguous Request Endpoint

Management List Ambiguous Request Endpoint

Management Database Endpoints

Management Compile HS Database Endpoint

Management Get HS DB Version Endpoint

Management Reload Management HS DB Endpoint

Management Reload Worker HS DB Endpoint

Management Reload Workers HS DB Endpoint

Management Kubernetes Endpoint

Management Get Management Pod Endpoint

Management Get Worker Pods Endpoint

Management Rules Endpoint

Management Add Rule Endpoint

Management Delete Rule Endpoint

Management Update Rule Endpoint

Management Get Rule Endpoint

Management Get Page Endpoint

Management Bulk Import Endpoint

Management Check Request Endpoint

Management Sync Endpoints

Management Sync Download Files Endpoint

Root Endpoints

Management UI Endpoint

Worker Redirect Endpoint

1.3 Search documentation

If you are looking for something specific try searching the documentation.

- search

Python Module Index

Index