Relé

Mercadona

Mar 04, 2024

CONTENTS

Release v1.15.0. (Installation)

Relé makes integration with Google PubSub easier and is ready to integrate seamlessly into any Django project.

The Publish-Subscribe pattern and specifically the Google Cloud Pub/Sub library are very powerful tools but you can easily cut your fingers on it. Relé makes integration seamless by providing Publisher, Subscriber and Worker classes.

CHAPTER

ONE

FEATURES

Out of the box, Relé includes the following features:

- Powerful Publishing API
- Highly Scalable Worker
- Intuitive Subscription Management
- Easily Extensible Middleware
- Ready to go Django/Flask integration
- CLI
- And much more…

CHAPTER

TWO

WHAT IT LOOKS LIKE

```python
# Subscribe to the Pub/Sub topic
from rele import sub
@sub(topic='photo-uploaded')
def photo_uploaded(data, **kwargs):
    print(f"Customer {data['customer_id']} has uploaded an image")

# Publish to the topic
import rele
rele.publish(topic='photo-uploaded', data={'customer_id': 123})
```

CHAPTER

THREE

INSTALL

Relé supports Python 3.6+ and installing via `pip`

```
$ pip install rele
```

or with Django integration

```
$ pip install rele[django,flask]
```

CHAPTER

FOUR

USER GUIDES

4.1 First Steps

4.1.1 Configuration

In order to get started using Relé, we must have a PubSub topic in which to publish. Via the Google Cloud Console we create one, named `photo-upload`.

To authenticate our publisher and subscriber, follow the Google guide on how to obtain your authentication account.

4.1.2 Publishing

To configure Relé, our settings may look something like:

```
# /settings.py

RELE = {
    'GC_CREDENTIALS_PATH': 'credentials.json',
}
```

```
# /publisher.py

import rele
import settings # we need this for initializing the global Publisher singleton

config = rele.config.setup(settings.RELE)
data = {
    'customer_id': 123,
    'location': '/google-bucket/photos/123.jpg'
}

rele.publish(topic='photo-uploaded', data=data)
```

To publish data, we simply pass in the topic to which we want our data to be published to, followed by a valid json serializable Python object.

Note: If you want to publish other types of objects, you may configure a custom *ENCODER_PATH*.

If you need to pass in additional attributes to the Message object, you can simply add `kwargs`. These must all be strings:

```
rele.publish(topic='photo-uploaded',
             data=data,
             type='profile',
             rotation='landscape')
```

Note: Anything other than a string attribute will result in a `TypeError`.

4.1.3 Subscribing

Once we can publish to a topic, we can subscribe to the topic from a worker instance. In an app directory, we create our sub function within our `subs.py` file.

```
# /app/subs.py

from rele import sub

@sub(topic='photo-uploaded')
def photo_uploaded(data, **kwargs):
    print(f"Customer {data['customer_id']} has uploaded an image to our service,
            and we stored it at {data['location'}.")
```

Additionally, if you added message attributes to your Message, you can access them via the `kwargs` argument:

```
@sub(topic='photo-uploaded')
def photo_uploaded(data, **kwargs):
    print(f"Customer {data['customer_id']} has uploaded an image to our service,
            and we stored it at {data['location'}.
            It is a {kwargs['type']} picture with the
            rotation {kwargs['rotation']}")
```

Message attributes

It might be helpful to access particular message attributes in your subscriber. One attribute that _rele_ adds by default is `published_at`. To access this attribute you can use `kwargs`.

```
@sub(topic='photo-uploaded')
def photo_uploaded(data, **kwargs):
    print(f"Customer {data['customer_id']} has uploaded an image to our service,
            and it was published at {kwargs['published_at'}.")
```

4.1.4 Consuming

Once the sub is implemented, we can start our worker which will register the subscriber on the topic with Google Cloud and will begin to pull the messages from the topic.

```
rele-cli run
```

In addition, if the `settings.py` module is not in the current directory, we can specify the path.

```
rele-cli run --settings app.settings
```

Note: Autodiscovery of subscribers with `rele-cli` is automatic. Any `subs.py` module you have in your current path, will be imported, and all subsequent decorated objects will be registered.

├──settings.py
├──app # This can be called whatever you like
├────subs.py

In another terminal session when we run `python publisher.py`, we should see the print readout in our subscriber.

4.2 Django Integration

Note: This guide simply points out the differences between standalone Relé and the Django integration. The basics about publishing and subscribing are described in the *First Steps* section.

4.2.1 Publishing

To configure Relé, our settings may look something like:

```
RELE = {
    'GC_CREDENTIALS_PATH': 'photo_project/settings/dummy-credentials.json',
    'MIDDLEWARE': [
        'rele.contrib.LoggingMiddleware',
        'rele.contrib.DjangoDBMiddleware',
    ],
    'APP_NAME': 'photo-imaging',
}
```

The only major difference here is that we are using the `rele.contrib.DjangoDBMiddleware`. This is important to properly close DB connections.

Important: If you plan on having your subscriber connect to the database, it is vital that the Django settings.CONN_MAX_AGE is set to 0.

Once the topic is created and our Django application has the proper configuration defined in *Settings*, we can start publishing to that topic.

4.2.2 Subscribing

Since the Django integration comes with `python manage.py runrele` command, we must name the file where we define our subscribers `subs.py`. `runrele` will auto-discover all decorated subscriber methods in a defined Django app and register/create the subscriptions for us.

Subscribing follows the same method as before.

4.2.3 Consuming

Unlike what is described in *Consuming*, the Django integration provides a very convenient command.

By running `python manage.py runrele`, worker process will autodiscover any properly decorated `@sub` function in the `subs.py` file and create the subscription for us.

Once the process is up and running, we can publish and consume.

4.3 Flask Integration

Note: This guide simply points out the differences between standalone Relé and the Flask integration. The basics about publishing and consuming are described in the *First Steps* section.

4.3.1 Setup

To configure Relé, our settings may look something like:

```
RELE = {
    'GC_CREDENTIALS_PATH': 'photo_project/settings/dummy-credentials.json',
    'MIDDLEWARE': [
        'rele.contrib.LoggingMiddleware',
        'rele.contrib.FlaskMiddleware',
    ],
    'APP_NAME': 'photo-imaging',
}

# Later when we setup rele and flask:
app = Flask()
rele.config.setup(RELE, flask_app=app)
```

The only major difference here is that we are using the `rele.contrib.FlaskMiddleware` and that we pass the Flask `app` instance to `rele.config.setup` method.

4.3.2 Subscribing

Now that that the middleware is setup our subscriptions will automatically have Flask's app context pushed when they are invoked so you will have access to the database connection pool and all other app dependent utilities.

```
from models import File
from database import db

@sub(topic='photo-uploads')
def handle_upload(data, **kwargs):
    new_file = File(data)
    db.session.add(new_file)
    db.session.commit()
```

4.4 Filtering Messages

Filter can be used to execute a subscription with specific parameters. There are three types of filters, global, by passing a filter_by parameter in the subscription (this applies the filter locally) or by passing a backend_filter_by parameter in the subscription (this applies the filter on pubsub).

4.4.1 *filter_by* parameter

This filter is a function that is supposed to return a boolean and this function is passed as parameter `filter_by` in the subscription.

```
def landscape_filter(kwargs):
    return kwargs.get('type') == 'landscape'


# This subscription is going to be called if in the kwargs
# has a key type with value landscape

@sub(topic='photo-updated', filter_by=landscape_filter)
def sub_process_landscape_photos(data, **kwargs):
    print(f'Received a photo of type {kwargs.get("type")}')
```

4.4.2 *backend_filter_by* parameter

This filter is an expression that is applied to the subscription creation. This filter expression is applied by pubsub before passing the message to the subscriber. More info about filter expressions here.

Note: Filter expressions are only applied on the subscription creation, they are not updated if changed if you do not recreate the subscription on pubsub.

```
# This subscription is going to be called if in the kwargs
# has a key type with value landscape

@sub(topic='photo-updated', backend_filter_by='attributes:type = "landscape"')
```

(continues on next page)

(continued from previous page)

```
def sub_process_landscape_photos(data, **kwargs):
    print(f'Received a photo of type {kwargs.get("type")}')
```

4.4.3 Global Filter

This filter is specified in the settings with the key `FILTER_SUBS_BY` that has a function as value. In case a subscription has a filter already it's going to use it's own filter.

```
import os

def landscape_filter(kwargs):
    return kwargs.get('type') == 'landscape'

settings = {
    ...
    'FILTER_SUBS_BY': landscape_filter,
}
```

4.5 Pub/Sub Emulator

It can be helpful to be able run the emulator in our development environment. To be able to do that we can follow the steps below:

1) Run the Google Cloud Pub/Sub emulator in the cloud-sdk container and map the port 8085.

```
$ docker pull google/cloud-sdk # Pull container
$ docker run -it --rm -p "8085:8085" google/cloud-sdk gcloud beta emulators pubsub start␣
↪--host-port=0.0.0.0:8085
```

2) Export PUBSUB_EMULATOR_HOST environment variable to specify the emulator host.

   In case you don't want to set this variable, it will be necessary to have pub/sub credentials.

```
$ export PUBSUB_EMULATOR_HOST=localhost:8085
```

3) Set rele settings in the Django project.

```
# my_django_project/settings.py

RELE = {
    'APP_NAME': 'my-awesome-app',
    'SUB_PREFIX': 'test',
    'GC_CREDENTIALS_PATH': 'my-credentials',
    'MIDDLEWARE': [
        'rele.contrib.LoggingMiddleware',
        'rele.contrib.DjangoDBMiddleware',
    ],
}
```

In case it's necessary to create a topic manually we can add it using the django shell.

```
python manage.py shell
```

```
from django.conf import settings
from google.cloud import pubsub_v1

publisher_client = pubsub_v1.PublisherClient()
topic_path = publisher_client.topic_path(settings.RELE.get('GC_PROJECT_ID'), 'topic_name
↪')
publisher_client.create_topic(topic_path)
```

4.6 Unrecoverable Middleware

To acknowledge and ignore incompatible messages that your subscription is unable to handle, you can use the *UnrecoverableMiddleware*.

4.6.1 Usage

First make sure the middleware is included in your Relé config.

```
# settings.py
import rele
from google.oauth2 import service_account

RELE = {
    'GC_CREDENTIALSGC_CREDENTIALS_PATH': 'credentials.json',
    'MIDDLEWARE': ['rele.contrib.UnrecoverableMiddleWare']
}
config = rele.config.setup(RELE)
```

Then in your subscription handler if you encounter an incompatible message raise the *UnrecoverableException*. Your message will be *.acked()* and it will not be redelivered to your subscription.

```
from rele.contrib.unrecoverable_middleware import UnrecoverableException
from rele import sub

@sub(topic='photo-uploaded')
def photo_uploaded(data, **kwargs):

  if data.get("required_property") is None:
      # Incompatible
      raise UnrecoverableException("required_property is required.")

  # Handle correct messages
```

CHAPTER

FIVE

CONFIGURATION

Here you can see the full list of the settings options for your deployment of Relé.

5.1 Settings

5.1.1 RELE

Default: `{}` (Empty dictionary)

A dictionary mapping all Relé configuration settings to values defined in your Django project's `settings.py`. Example:

```
RELE = {
    'GC_CREDENTIALS_PATH': 'rele/settings/dummy-credentials.json',
    'MIDDLEWARE': [
```

(continues on next page)

(continued from previous page)

```
        'rele.contrib.LoggingMiddleware',
        'rele.contrib.DjangoDBMiddleware',
    ],
    'SUB_PREFIX': 'mysubprefix',
    'APP_NAME': 'myappname',
    'ENCODER_PATH': 'rest_framework.utils.encoders.JSONEncoder',
    'ACK_DEADLINE': 120,
    'PUBLISHER_TIMEOUT': 3.0,
    'FILTER_SUBS_BY': boolean_function,
    'DEFAULT_RETRY_POLICY': RetryPolicy(10, 50),
    'GC_STORAGE_REGION': 'europe-west1',
    'CLIENT_OPTIONS': {'api_endpoint': 'custom-api.interconnect.example.com'}
}
```

5.1.2 GC_PROJECT_ID

Optional

GCP project id to use. If this is not provided then it is inferred via either service account's project id or quota project id if using Application Default Credentials (ADC)

5.1.3 GC_CREDENTIALS_PATH

Optional

Path to service account json file with access to PubSub

5.1.4 MIDDLEWARE

Optional

Default: `['rele.contrib.LoggingMiddleware']`

List of the middleware modules that will be included in the project. The order of execution follows FIFO.

It is strongly recommended that for Django integration, you add:

```
[
    'rele.contrib.LoggingMiddleware',
    'rele.contrib.DjangoDBMiddleware',
]
```

The DjangoDBMiddleware will take care of opening and closing connections to the db before and after your callbacks are executed. If this is left out, it is highly probable that your database will run out of connections in your connection pool.

The LoggingMiddleware will take care of logging subscription information before and after the callback is executed. The subscription message is only logged when an exception was raised while processing it. If you would like to log this message in every case, you should create a middleware of your own.

5.1.5 SUB_PREFIX

Optional

A prefix to all your subs that can be declared globally.

For instance, if you have two projects listening to one topic, you may want to add a prefix so that there can be two distinct subscribers to that one topic.

5.1.6 APP_NAME

Optional

The application name.

This should be unique to all the services running in the application ecosystem. It is used by the LoggingMiddleware and Prometheus integration.

5.1.7 ENCODER_PATH

Optional

Default: rest_framework.utils.encoders.JSONEncoder

Encoder class path to use for serializing your Python data structure to a json object when publishing.

Note: The default encoder class is subject to change in an upcoming release. It is advised that you use this setting explicitly.

5.1.8 ACK_DEADLINE

Optional

Ack deadline for all subscribers in seconds.

See also:

The Google Pub/Sub documentation which states that *The subscriber has a configurable, limited amount of time – known as the ackDeadline – to acknowledge the outstanding message. Once the deadline passes, the message is no longer considered outstanding, and Cloud Pub/Sub will attempt to redeliver the message.*

5.1.9 PUBLISHER_BLOCKING

Optional

Default: False

Wait synchronously for the publishing result

See Google PubSub documentation for more info

5.1.10 PUBLISHER_TIMEOUT

Optional

Default: 3.0 seconds

Timeout that the publishing result will wait on the future to publish successfully while blocking.

See Google PubSub documentation for more info

5.1.11 THREADS_PER_SUBSCRIPTION

Optional

Default: 2

Number of threads that will be consumed for each subscription. Default behavior of the Google Cloud PubSub library is to use 10 threads per subscription. We thought this was too much for a default setting and have taken the liberty of reducing the thread count to 2. If you would like to maintain the default Google PubSub library behavior, please set this value to 10.

5.1.12 FILTER_SUBS_BY

Optional

Boolean function that applies a global filter on all subscriptions. For more information, please see Filtering Messages section.

5.1.13 DEFAULT_RETRY_POLICY

Optional

A RetryPolicy object which must be instantiated with *minimum_backoff* and *maximum_backoff*, that specifies in seconds how Pub/Sub retries message delivery for all the subscriptions.

If not set, the default retry policy is applied, meaning a minimum backoff of 10 seconds and a maximum backoff of 60 seconds. This generally implies that messages will be retried as soon as possible for healthy subscribers. RetryPolicy will be triggered on NACKs or acknowledgement deadline exceeded events for a given message.

5.1.14 GC_STORAGE_REGION

Optional

Set the Google Cloud's region for storing the messages. By default is *europe-west1*

5.1.15 CLIENT_OPTIONS

Optional

Provide custom options for publisher and subscriber client. Following are three of the options.

1. The *api_endpoint* property can be used to override the default endpoint provided by the client when transport is not explicitly provided.
2. The *client_cert_source* property can be used to provide a client certificate for mTLS transport. If not provided, the default SSL client certificate will be used if present.
3. The *universe_domain* property can be used to override the default "googleapis.com" universe. Note that the api_endpoint property still takes precedence; and universe_domain is currently not supported for mTLS.

For more information about the client options, please see Publisher Client and Subscriber Client.

CHAPTER

SIX

API DOCS

This is the part of documentation that details the inner workings of Relé. If you are looking for information on a specific function, class or method, this part of the documentation is for you.

6.1 API Reference

6.1.1 Clients

class rele.client.**Publisher**(*gc_project_id*, *credentials*, *encoder*, *timeout*, *client_options*, *blocking=None*)

The Publisher Class

Wraps the Google Cloud Publisher Client and handles encoding of the data.

It is important that this class remains a Singleton class in the process. Otherwise, a memory leak will occur. To avoid this, it is strongly recommended to use the *publish()* method.

If the setting *USE_EMULATOR* evaluates to True, the Publisher Client will not have any credentials assigned.

Parameters

- **gc_project_id** – string Google Cloud Project ID.
- **credentials** – string Google Cloud Credentials.
- **encoder** – A valid json.encoder.JSONEncoder subclass # noqa
- **timeout** – float, default *PUBLISHER_TIMEOUT*
- **blocking** – boolean, default None falls back to *PUBLISHER_BLOCKING*

publish(*topic*, *data*, *blocking=None*, *timeout=None*, *raise_exception=True*, ***attrs*)

Publishes message to Google PubSub topic.

Usage:

```
publisher = Publisher()
publisher.publish('topic_name', {'foo': 'bar'})
```

By default, this method is non-blocking, meaning that the method does not wait for the future to be returned.

If you would like to wait for the future so you can track the message later, you can:

Usage:

```
publisher = Publisher()
future = publisher.publish('topic_name', {'foo': 'bar'}, blocking=True,␣
↪timeout=10.0) # noqa
```

However, it should be noted that using *blocking=True* may incur a significant performance hit.

In addition, the method adds a timestamp *published_at* to the message attrs using epoch floating point number.

Parameters

- **topic** – string topic to publish the data.
- **data** – dict with the content of the message.
- **blocking** – boolean, default None falls back to *PUBLISHER_BLOCKING*
- **timeout** – float, default None falls back to *PUBLISHER_TIMEOUT*
- **raise_exception** – boolean. If True, exceptions coming from PubSub will be raised
- **attrs** – additional string parameters to be published.

Returns

Future # noqa

class rele.client.**Subscriber**(*gc_project_id*, *credentials*, *message_storage_policy*, *client_options*, *default_ack_deadline=None*, *default_retry_policy=None*)

The Subscriber Class.

For convenience, this class wraps the creation and consumption of a topic subscription.

Parameters

- **gc_project_id** – str *MIDDLEWARE* .
- **credentials** – obj `credentials()`.
- **message_storage_policy** – str Region to store the messages
- **default_ack_deadline** – int Ack Deadline defined in settings
- **default_retry_policy** – RetryPolicy Rele's RetryPolicy defined in settings

consume(*subscription_name*, *callback*, *scheduler*)

Begin listening to topic from the SubscriberClient.

Parameters

- **subscription_name** – str Subscription name
- **callback** – Function which act on a topic message
- **scheduler** – Thread pool-based scheduler. # noqa

Returns

Future # noqa

update_or_create_subscription(*subscription*)

Handles creating the subscription when it does not exists or updates it if the subscription contains any parameter that allows it.

This makes it easier to deploy a worker and forget about the subscription side of things. If the topic of the subscription do not exist, it will be created automatically.

Parameters
subscription – obj *Subscription*.

6.1.2 Publish

rele.publishing.**publish**(*topic*, *data*, ***kwargs*)

Shortcut method to publishing data to PubSub.

This is a shortcut method that instantiates the Publisher if not already instantiated in the process. This is to ensure that the Publisher remains a Singleton class.

Usage:

```
import rele

def myfunc():
    # ...
    rele.publish(topic='lets-tell-everyone',
                 data={'foo': 'bar'},
                 myevent='arrival')
```

Parameters

- **topic** – str PubSub topic name
- **data** – dict-like Data to be sent as the message.
- **timeout** – float. Default None, falls back to RELE['PUBLISHER_TIMEOUT'] value
- **blocking** – boolean. Default False
- **kwargs** – Any optional key-value pairs that are included as attributes in the message

Returns
None

6.1.3 Subscription

class rele.subscription.**Subscription**(*func*, *topic*, *prefix=''*, *suffix=''*, *filter_by=None*, *backend_filter_by=None*, *retry_policy=None*)

The Subscription class

In addition to using the `@sub` decorator, it is possible to subclass the Subscription.

For example:

```
from rele import Subscription

class DoSomethingSub(Subscription):
    topic = 'photo-uploaded'

    def __init__(self):
        self._func = self.callback_func
        super().__init__(self._func, self.topic)
```

(continues on next page)

(continued from previous page)

```
    def callback_func(self, data, **kwargs):
        print(data["id"])
```

If `rele-cli run` is used, the `DoSomethingSub` will be a valid subscription and registered on Google Cloud.

`rele.subscription.`**`sub`**(*topic*, *prefix=None*, *suffix=None*, *filter_by=None*, *backend_filter_by=None*, *retry_policy=None*)

Decorator function that makes declaring a PubSub Subscription simple.

The Subscriber returned will automatically create and name the subscription for the topic. The subscription name will be the topic name prefixed by the project name.

For example, if the topic name to subscribe too is *lets-tell-everyone*, the subscriber will be named *project-name-lets-tell-everyone*.

Additionally, if a *suffix* param is added, the subscriber will be *project-name-lets-tell-everyone-my-suffix*.

It is recommended to add ***kwargs* to your *sub* function. This will allow message attributes to be sent without breaking the subscriber implementation.

Usage:

```
@sub(topic='lets-tell-to-alice', prefix='shop')
def bob_purpose(data, **kwargs):
     pass

@sub(topic='lets-tell-everyone', suffix='sub1')
def purpose_1(data, **kwargs):
     pass

@sub(topic='lets-tell-everyone', suffix='sub2')
def purpose_2(data, **kwargs):
     pass

@sub(topic='photo-updated',
     filter_by=lambda **attrs: attrs.get('type') == 'landscape')
def sub_process_landscape_photos(data, **kwargs):
    pass
```

Parameters

- **`topic`** – string The topic that is being subscribed to.
- **`prefix`** – string An optional prefix to the subscription name. Useful to namespace your subscription with your project name
- **`suffix`** – string An optional suffix to the subscription name. Useful when you have two subscribers in the same project that are subscribed to the same topic.
- **`filter_by`** – Union[function, list] An optional function or tuple of functions that filters the messages to be processed by the sub regarding their attributes.
- **`retry_policy`** – obj `RetryPolicy`

Returns

Subscription

6.1.4 Worker

exception rele.worker.**NotConnectionError**

class rele.worker.**Worker**(*subscriptions*, *client_options*, *gc_project_id=None*, *credentials=None*, *gc_storage_region=None*, *default_ack_deadline=None*, *threads_per_subscription=None*, *default_retry_policy=None*)

A Worker manages the subscriptions which consume Google PubSub messages.

Facilitates the creation of subscriptions if not already created, and the starting and stopping the consumption of them.

Parameters
subscriptions – list *Subscription*

run_forever(*sleep_interval=1*)

Shortcut for calling setup, start, and _wait_forever.

Parameters
sleep_interval – Number of seconds to sleep in the `while True` loop

setup()

Create the subscriptions on a Google PubSub topic.

If the subscription already exists, the subscription will not be re-created. Therefore, it is idempotent.

start()

Begin consuming all subscriptions.

When consuming a subscription, a `StreamingPullFuture` is returned from the Google PubSub client library. This future can be used to manage the background stream.

The futures are stored so that they can be cancelled later on for a graceful shutdown of the worker.

stop(*signal=None*, *frame=None*)

Manage the shutdown process of the worker.

This function has two purposes:

1. Cancel all the futures created.
2. And close all the database connections opened by Django. Even though we cancel the connections for every execution of the callback, we want to be sure that all the database connections are closed in this process.

Exits with code 0 for a clean exit.

Parameters

- **signal** – Needed for signal.signal # noqa
- **frame** – Needed for signal.signal # noqa

rele.worker.**create_and_run**(*subs*, *config*)

Create and run a worker from a list of Subscription objects and a config while waiting forever, until the process is stopped.

We stop a worker process on: - SIGINT - SIGTSTP

Parameters

- **subs** – List *Subscription*
- **config** – `Config`

6.1.5 Middleware

Relé middleware's provide additional functionality to default behavior. Simply subclass `BaseMiddleware` and declare the hooks you wish to use.

6.1.6 Base Middleware

class rele.middleware.**BaseMiddleware**

Base class for middleware. The default implementations for all hooks are no-ops and subclasses may implement whatever subset of hooks they like.

post_process_message()

Called after the Worker processes the message.

post_process_message_failure(*subscription*, *exception*, *start_time*, *message*)

Called after the message has been unsuccessfully processed. :param subscription: :param exception: :param start_time: :param message:

post_process_message_success(*subscription*, *start_time*, *message*)

Called after the message has been successfully processed. :param subscription: :param start_time: :param message:

post_publish_failure(*topic*, *exception*, *message*)

Called after publishing fails. :param topic: :param exception: :param message:

post_publish_success(*topic*, *data*, *attrs*)

Called after Publisher succesfully sends message. :param topic: :param data: :param attrs:

post_worker_start()

Called after the Worker process starts up.

post_worker_stop()

Called after the Worker process shuts down.

pre_process_message(*subscription*, *message*)

Called when the Worker receives a message. :param subscription: :param message:

pre_publish(*topic*, *data*, *attrs*)

Called before Publisher sends message. :param topic: :param data: :param attrs:

pre_worker_start()

Called before the Worker process starts up.

pre_worker_stop(*subscriptions*)

Called before the Worker process shuts down.

setup(*config*, ***kwargs*)

Called when middleware is registered. :param config: Relé Config object

6.1.7 Logging Middleware

class rele.contrib.logging_middleware.**LoggingMiddleware**

Default logging middleware.

Logging format has been configured for Prometheus.

post_process_message_failure(*subscription*, *exception*, *start_time*, *message*)

Called after the message has been unsuccessfully processed. :param subscription: :param exception: :param start_time: :param message:

post_process_message_success(*subscription*, *start_time*, *message*)

Called after the message has been successfully processed. :param subscription: :param start_time: :param message:

post_publish_failure(*topic*, *exception*, *message*)

Called after publishing fails. :param topic: :param exception: :param message:

post_publish_success(*topic*, *data*, *attrs*)

Called after Publisher succesfully sends message. :param topic: :param data: :param attrs:

pre_process_message(*subscription*, *message*)

Called when the Worker receives a message. :param subscription: :param message:

pre_publish(*topic*, *data*, *attrs*)

Called before Publisher sends message. :param topic: :param data: :param attrs:

pre_worker_stop(*subscriptions*)

Called before the Worker process shuts down.

setup(*config*, ***kwargs*)

Called when middleware is registered. :param config: Relé Config object

6.1.8 Django Middleware

6.2 Changelog

Here you can see the full list of changes between each Relé release.

6.2.1 Changelog

1.15.0 (2024-02-29)

- [Added] Add a setting CLIENT_OPTIONS for Pub/Sub clients (#274)
- [Changed] Use api_endpoint from CLIENT_OPTIONS to check loss of network connectivity(#275)

1.14.0 (2024-02-01)

- [Fixed] Restart worker when get in an inconsistent status due a loss of network connectivity (#267)
- [Added] Allow –third-party-subscriptions argument in rele-cli run command for stand alone workers (#269)

1.13.0 (2023-09-04)

- [Added] Add verbosity to *VerboseLoggingMiddleware*'s hooks (#240)

1.12.0 (2023-07-17)

- [Added] Check if subs have same memory address (#257)
- [Changed] Detect subs module at any folder level (#255)

1.11.0 (2023-05-09)

- [Added] Allow updating retry policy to existing subscriptions. (#248)

1.10.0 (2023-05-02)

- [Added] Add configuration for setting the storage region for pubsub messages (#247)

1.9.0 (2023-05-02)

- [Changed] Use custom encoder in logging middleware. (#247)

1.8.0 (2023-04-28)

- [Added] Add retry policy to subscriptions. (#222)

1.7.0 (2022-11-15)

- [Added] Add PUBLISHER_BLOCKING setting
- [Changed] Provide a *subscription_message* argument of a consistent data type to all hooks
- [Changed] Fix rendering of links in docs
- [Changed] Add improvements for local development

1.6.0 (2022-08-03)

- [Added] Implement auto restart of the consumption when futures are done or cancelled. (#226)

1.5.0 (2022-04-20)

- [Added] Add filter expressions to subscriptions. (#207)

1.4.1 (2022-04-19)

- [Modified] Fixed bug in the post-publish-failure VerboseLoggingMiddleware hook. (#220)

1.4.0 (2022-04-13)

- [Added] Added a VerboseLoggingMiddleware that does not truncate mesage payload. (#218)

1.3.0 (2022-04-04)

- GC Project Id & Windows support (#215)

1.2.0 (2021-12-10)

- [CHANGED] TimeotError from publisher (#212)
- Added filter_subs_by setting in documentation (#208)
- Automatic topic creation (#206)
- Log post publish success (#204)

1.1.1 (2021-6-28)

- Do not define default_app_config, it's deprecated (#199)
- Do not implement deprecated middlewares in the base class (#200)

1.1.0 (2021-3-10)

- Google Pubsub 2.0 Compat (#192)
- Add validations to the sub decorator (#189)
- Add new post_publish_hook and deprecate the old one (#190)
- Discover and load settings when publishing (#188)
- Fix #180: Raise error when the config loads a repeated subscription (#187)

1.0.0 (2020-9-25)

- BREAKING: Remove GC_PROJECT_ID (#183)

0.14.0 (2020-8-5)

- BREAKING: Remove GC_CREDENTIALS (#174)
- Add changelog to the docs site (#179)
- Catch TimeoutError and run post_publish_failure when blocking (#172)
- Deprecate GC_PROJECT_ID setting (#178)

0.13.0 (2020-7-9)

- Add documentation for class based subscriptions (#169)
- Deprecate GC_CREDENTIALS setting (#173)
- GC_CREDENTIALS_PATH setting option (#170)

0.13.dev0 (2020-6-16)

- Traverse all packages to autodiscover all subs.py modules (#167)
- Auto-discovery of class based subscriptions (#168)

0.12.0 (2020-6-12)

- Added `--settings` path option in CLI (#166)
- Added isort linting (#164)

0.11.0 (2020-6-4)

- CLI feature (#160)
- Documentation Enhancements (#158, #155, #162)
- Testing Improvements (#154, #157)

0.10.0 (2020-2-4)

- Adjust default THREADS_PER_SUBSCRIPTION (#152)
- Add unrecoverable_middleware (#150)
- Allow multiple filters (#148)
- Configure timeout from .publish() (#143)
- Dont crash when subscription topic does not exist (#142)

0.9.1 (2020-1-2)

- Ack messages when data not json serializable (#141)
- Use ThreadScheduler instead of ThreadPoolExecutor (#145)

0.9.0 (2019-12-20)

- Flask support via middleware (#127)
- Add message attributes to metrics log (#128)
- Specify number of threads per subscriber with Subscription ThreadPoolExecutor (#139)
- Publishing timeout while blocking (#137)
- Clean up rele.config.setup + Worker() init (#132)

0.8.1 (2019-11-25)

- Fix runrele command

0.8.0 (2019-11-22)

- Worker run method (#118)
- Add kwargs to setup method passed through to middleware (#123)
- Add missing worker middleware hooks (#121)
- Add 3.8 support
- More Documentation

0.7.0 (2019-10-21)

- BREAKING: Remove Django as a dependency (#95)
- More documentation

0.6.0 (2019-09-21)

- BREAKING: Remove drf as a dependency (#91)
- Add message as a parameter for middleware hooks (#99)
- Check setting.CONN_MAX_AGE and warn when not 0 (#97)
- More documentation

0.5.0 (2019-08-08)

- `python manage.py showsubscriptions` command
- Configurable ENCODER setting
- Move DEFAULT_ACK_DEADLINE to the RELE config
- More documentation

0.4.1 (2019-06-18)

- Ability to install app only with rele
- Define default filter_by in settings.RELE

0.4.0 (2019-06-17)

- Set `DEFAULT_ACK_DEADLINE` (#49)
- Filter by message attributes (#66)
- BREAKING: All Relé settings are defined in a dict (#60)

Old structure:

```
from google.oauth2 import service_account
RELE_GC_CREDENTIALS = service_account.Credentials.from_service_account_file(
    'rele/settings/dummy-credentials.json'
)
RELE_GC_PROJECT_ID = 'dummy-project-id'
```

New structure:

```
from google.oauth2 import service_account
RELE = {
    'GC_CREDENTIALS': service_account.Credentials.from_service_account_file(
        'rele/settings/dummy-credentials.json'
    ),
    'GC_PROJECT_ID': 'dummy-project-id',
    'MIDDLEWARE': [
        'rele.contrib.LoggingMiddleware',
        'rele.contrib.DjangoDBMiddleware',
    ],
    'SUB_PREFIX': 'mysubprefix',
    'APP_NAME': 'myappname',
}
```

- `rele.contrib.middleware` (#55)
- Prefix argument in sub decorator (#47)
- Add timestamp to the published message (#42)
- BREAKING: Explicit publisher and subscriber configuration (#43)
- Sphinx documentation (#27, #34, #40, #41)
- Contributing guidelines (#32)

0.3.1 (2019-06-04)

- Add prometheus metrics key to logs (#16 - #20, #22, #23)
- Fix JSON serialization when publishing (#25)

0.3.0 (2019-05-14)

- Ability to run in emulator mode (#12)
- Add Travis-CI builds (#10)
- More friendly global publish (#11)
- Non-blocking behaviour when publishing by default (#6)

0.2.0 (2019-05-09)

- Initial version

CHAPTER

SEVEN

PROJECT INFO

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

S

U

W