Hedwig Documentation

Release 8.9.1-dev

Aniruddha Maru

Jun 25, 2021

CONTENTS

Hedwig is a inter-service communication bus that works on AWS SQS/SNS, while keeping things pretty simple and straight forward. It uses json schema draft v4 for schema validation so all incoming and outgoing messages are validated against pre-defined schema.

Hedwig allows separation of concerns between consumers and publishers so your services are loosely coupled, and the contract is enforced by the schema validation. Hedwig may also be used to build asynchronous APIs.

For intra-service messaging, see Taskhawk.

Only Python 3.6+ is supported currently.

This project uses semantic versioning.

CHAPTER

ONE

QUICKSTART

1.1 Quickstart

Getting started with Hedwig is easy, but requires a few steps.

1.1.1 Installation

Install the latest hedwig release via *pip*:

```
$ pip install authedwig[aws,jsonschema]
```

If using Google, use `authedwig[gcp,jsonschema]`.

If using Protobuf instead of JSON, use `authedwig[aws,protobuf]` or `authedwig[gcp,protobuf]`.

You may also install a specific version:

```
$ pip install authedwig[aws,jsonschema]==1.0.0
```

The latest development version can always be found on Github.

1.1.2 Configuration

Before you can use Hedwig, you need to set up a few settings. For Django projects, simple use Django settings to configure Hedwig. For Flask projects, use Flask config. For other frameworks, you can either declare an environment variable called `SETTINGS_MODULE` that points to a module where settings may be found, or manually configure using `hedwig.conf.settings.configure_with_object`.

There are 2 cloud platforms currently supported: AWS and Google Cloud Platform. Settings will defer depending on your platform.

Common required settings:

```
HEDWIG_CALLBACKS = <YOUR CALLBACKS>

HEDWIG_QUEUE = <YOUR APP HEDWIG QUEUE>

HEDWIG_MESSAGE_ROUTING = <YOUR INFRA ROUTES>

HEDWIG_JSONSCHEMA_FILE = <PATH TO YOUR SCHEMA FILE>
```

When using AWS, additional required settings are:

```
AWS_ACCESS_KEY = <YOUR AWS KEY>
AWS_ACCOUNT_ID = <YOUR AWS ACCOUNT ID>
AWS_REGION = <YOUR AWS REGION>
AWS_SECRET_KEY = <YOUR AWS SECRET KEY>

HEDWIG_CONSUMER_BACKEND = 'hedwig.backends.aws.AWSSQSConsumerBackend'
HEDWIG_PUBLISHER_BACKEND = 'hedwig.backends.aws.AWSSNSPublisherBackend'
```

In case of GCP, additional required settings are:

```
HEDWIG_CONSUMER_BACKEND = 'hedwig.backends.gcp.GooglePubSubConsumerBackend'
HEDWIG_PUBLISHER_BACKEND = 'hedwig.backends.gcp.GooglePubSubPublisherBackend'

HEDWIG_SUBSCRIPTIONS = <LIST OF YOUR HEDWIG TOPIC NAMES APP IS SUBSCRIBED TO>
```

If running outside Google Cloud (e.g. locally), set `GOOGLE_APPLICATION_CREDENTIALS`.

Within Google Cloud, these credentials and permissions are managed by Google using IAM.

If the Pub/Sub resources lie in a different project, set `GOOGLE_CLOUD_PROJECT` to the project id.

For batch publish, use `hedwig.backends.gcp.GooglePubSubAsyncPublisherBackend`

1.1.3 Provisioning

Hedwig works on topics, queues, and subscriptions on AWS and Google cloud platforms. Before you can publish/consume messages, you need to provision the required infra. This may be done manually, or, preferably, using Terraform. Hedwig provides tools to make infra configuration easier: see Terraform and Hedwig Terraform Generator for further details.

Instrumentation

This library supports OpenTelemetry for application tracing. Headers are used to receive and publish trace contexts. Vendor specific tracing mechanisms (e.g. AWS X-Ray) are currently not supported. Vendor specific formats however are supported by customization of *set_global_textmap* (e.g. GCP *X-Cloud-Trace-Context*).

1.1.4 Fan Out

Hedwig utilizes SNS and Pub/Sub for fan-out configuration. A publisher publishes messages on a *topic*. This message may be received by zero or more consumers. The publisher needn't be aware of the consuming application at all. There are a variety of messages that may be published as such, but they generally fall into 2 buckets:

1. **Asynchronous API Requests**: Hedwig may be used to call APIs asynchronously. The contract is enforced by your infra-structure by connecting SNS topics to SQS queues, and payload is validated using the schema you define. Response is a delivered using a separate message if required.
2. **Notifications**: The most common use case is to notify other services/apps that may be interested in events. For example, your User Management app can publish a `user.created` message notification to all your apps. As publishers and consumers are loosely coupled, this separation of concerns is very effective in ensuring a stable eco-system.

1.1.5 Using Hedwig

To use hedwig, simply add a message handler like so:

```
def send_email(message: hedwig.models.Message) -> None:
    # send email
```

And then send a message:

```
message = hedwig.models.Message.new(
    "send_email",
    StrictVersion('1.0'),
    {
        'to': 'example@email.com',
        'subject': 'Hello!',
    },
)
message.publish()
```

Messages are held in SQS queue, or Pub/Sub Subscription until they're successfully executed, or until they fail a configurable number of times. Failed tasks are moved to a Dead Letter Queue, where they're held for 14 days, and may be examined for further debugging.

CHAPTER

TWO

USAGE

2.1 Usage Guide

2.1.1 Callbacks

Callbacks are simple python functions that accept a single argument of type `hedwig.models.Message` -

```
def send_email(message: hedwig.models.Message) -> None:
    # send email
```

You can access the data dict using `message.data` as well as custom headers using `message.headers` and other metadata fields as described in the API docs: *hedwig.models.Message()*.

2.1.2 Publisher

You can run publish messages like so:

```
models.Message.new("message.type", StrictVersion('1.0'), data).publish()
```

If you want to include a custom headers with the message (for example, you can include a `request_id` field for cross-application tracing), you can pass in additional parameter `headers`.

2.1.3 Consumer

A consumer for AWS SQS/Google PubSub based workers can be started as following:

```
consumer.listen_for_messages()
```

This is a blocking function. Don't use threads since this library is **NOT** guaranteed to be thread-safe.

A consumer for Lambda based workers can be started as following:

```
consumer.process_messages_for_lambda_consumer(lambda_event)
```

where `lambda_event` is the event provided by AWS to your Lambda function as described in lambda sns format.

2.1.4 Schema

JSON-Schema

The schema file must be a JSON-Schema draft v4 schema. There's a few more restrictions in addition to being a valid schema:

- There must be a top-level key called `schemas`. The value must be an object.
- `schemas`: The keys of this object must be message types. The value must be an object.
- `schemas/<message_type>`: The keys of this object must be major version patterns for this message type. The value must be an object.
- `schemas/<message_type>/<major_version>.*`: This object must represent the data schema for given message type, and major version. Any minor version updates must be applied in-place, and must be non-breaking per semantic versioning.

Note that the schema file only contains definitions for major versions. This is by design since minor version MUST be backwards compatible.

Optionally, a key `x-versions` may be used to list full versions under a major version.

For an example, see example jsonschema schema.

Protobuf

The proto file must be a proto3 schema and must be pre-compiled by the application. There's a few more restrictions in addition to being a valid schema:

- `<message_type>V<major_version>`: For every message type and every major version for that message type, a protobuf message with this name must be defined.
- Every protobuf message must include options `(hedwig.message_options).major_version` and `(hedwig.message_options).minor_version`.
- Multiple protobuf files for organizing the schemas is fine, but the final compiled version must be present as a single python module.

Note that the schema file only contains definitions for major versions. This is by design since minor version MUST be backwards compatible.

For an example, see example protobuf schema.

2.1.5 Testing

Hedwig supports pytest by default and provides pytest testing utilities as part of the *hedwig.testing.pytest_plugin* module.

2.2 Configuration

Add appropriate configuration to the app.

AWS_REGION

AWS region

required; string; AWS only

AWS_ACCOUNT_ID

AWS account id

required; string; AWS only

AWS_ACCESS_KEY

AWS access key

required; string; AWS only

AWS_CONNECT_TIMEOUT_S

AWS connection timeout

optional; int; default: 2; AWS only

AWS_ENDPOINT_SNS

AWS endpoint for SNS. This may be used to customized AWS endpoints to assist with testing, for example, using localstack.

optional; string; AWS only

AWS_ENDPOINT_SQS

AWS endpoint for SQS. This may be used to customized AWS endpoints to assist with testing, for example, using localstack.

optional; string; AWS only

AWS_READ_TIMEOUT_S

AWS read timeout

optional; int; default: 2; AWS only

AWS_SECRET_KEY

AWS secret key

required; string; AWS only

AWS_SESSION_TOKEN

AWS session token that represents temporary credentials (for example, for Lambda apps)

optional; string; AWS only

GOOGLE_APPLICATION_CREDENTIALS

Path to the Google application credentials json file. If running in Google Cloud, these is automatically managed by Google. If passed in explicitly, this is respected, and takes precedence. See Google Cloud Auth docs for further details.

optional; string; Google only

GOOGLE_CLOUD_PROJECT

The Google Project ID that contains Pub/Sub resources. This is automatically interpreted based on the credentials, however it may be overridden if Pub/Sub resources live in a different project.

optional; string; Google only

GOOGLE_PUBSUB_READ_TIMEOUT_S

Read from PubSub subscription timeout in seconds

optional: int; default: 5; Google only

HEDWIG_CALLBACKS

A dict of Hedwig callbacks, with values as callables or fully-qualified function names. The key is a tuple of message type and major version pattern of the schema.

required for consumers; `dict[tuple[string, string], string]`

HEDWIG_CONSUMER_BACKEND

Hedwig consumer backend class

required; string

HEDWIG_DATA_VALIDATOR_CLASS

The validator class to use for schema validation. This class must be a sub-class of `hedwig.validators.HedwigBaseValidator`, and may add additional validation logic. This class is also responsible for serialization / deserialization of the payload on the wire.

Validators provided by the library: jsonschema, protobuf, protobuf-json.

To customize jsonschema validator, for example, to add a new format called `vin`, use this validator:

```
class CustomValidator(hedwig.validators.JSONSchemaValidator):
    # simplistic check: 17 alphanumeric characters except i, o, q
    _vin_re = re.compile("^[a-hj-npr-z0-9]{17}$")

    @staticmethod
    @hedwig.validators.JSONSchemaValidator.checker.checks('vin')
    def check_vin(instance) -> bool:
        if not isinstance(instance, str):
            return True
        return bool(CustomValidator._vin_re.match(instance))
```

optional; fully-qualified class name; defaults to "hedwig.validators.jsonschema.JSONSchemaValidator"

HEDWIG_DEFAULT_HEADERS

A function that may be used to inject custom headers into every message, for example, request id. This hook is called right before dispatch, and any headers that are explicitly specified when dispatching may override these headers.

If specified, it's called with the following arguments:

```
default_headers(message=message)
```

where `message` is the outgoing Message object, and its expected to return a dict of strings.

It's recommended that this function be declared with `**kwargs` so it doesn't break on new versions of the library.

optional; fully-qualified function name

HEDWIG_MESSAGE_ROUTING

A dict of Hedwig message types, with values as topic names. The key is a tuple of message type and major version pattern of the schema. An entry is required for every message type that the app wants to publish. For publishing cross-project topic messages, instead of topic name, use: - AWS - a tuple of topic name and AWS account id (must exist in the same region) - Google - a tuple of topic name and GCP project id

It's recommended that major versions of a message be published on separate topics.

required; `dict[tuple[string, string], Union[string, Tuple[string, string]]]`

HEDWIG_PRE_PROCESS_HOOK

A function which can used to plug into the message processing pipeline *before* any processing happens. This hook may be used to perform initializations such as set up a global request id based on message headers. If specified, this will be called with the following arguments for AWS SQS apps:

```
pre_process_hook(sqs_queue_message=sqs_queue_message)
```

where `sqs_queue_message` is of type `boto3.sqs.Message`.

For Lambda apps as so:

```
pre_process_hook(sns_record=record)
```

where `sns_record` is a `dict` of a single record with format as described in lambda sns format.

For Google apps as so:

```
pre_process_hook(google_pubsub_message=google_pubsub_message)
```

where `google_pubsub_message` is of type `google.cloud.pubsub_v1.subscriber.message.Message`.

It's recommended that this function be declared with `**kwargs` so it doesn't break on new versions of the library.

optional; fully-qualified function name

HEDWIG_PROTOBUF_SCHEMA_MODULE

The name of the module representing the Hedwig protobuf schema. This module must be pre-compiled and must contain all messages. Each message type's schema must include all valid major versions.

required if using protobuf; string; fully-qualified module path

HEDWIG_POST_PROCESS_HOOK

Same as `HEDWIG_PRE_PROCESS_HOOK` but executed after message processing.

HEDWIG_PUBLISHER

Name of the publisher

required for publishers; string

HEDWIG_PUBLISHER_BACKEND

Hedwig publisher backend class

required; string

HEDWIG_PUBLISHER_GCP_BATCH_SETTINGS

Batching configuration for the `GooglePubSubAsyncPublisherBackend` publisher.

See Google PubSub Docs for more information.

optional; `google.cloud.pubsub_v1.BatchSettings`; Google only

HEDWIG_QUEUE

The name of the hedwig queue (exclude the `HEDWIG-` prefix).

required; string

HEDWIG_JSONSCHEMA_FILE

The filepath to a JSON-Schema file representing the Hedwig schema. This json-schema must contain all messages under a top-level key `schemas`. Each message's schema must include all valid versions for that message.

required if using json schema; string; filepath

HEDWIG_SUBSCRIPTIONS

List of all the Hedwig topics that the app is subscribed to (exclude the `hedwig-` prefix). For subscribing to cross-project topic messages, instead of topic name, use a tuple of topic name and GCP project id.

required; List(Union(string, Tuple[string, string])); Google only

HEDWIG_SYNC

Flag indicating if Hedwig should work synchronously. If set to `True` a published message will be dispatched immediately using `HEDWIG_CALLBACKS` without calling any SQS APIs. This is similar to Celery's Eager mode and is helpful for integration testing. It's assumed that your service handles the message you're dispatching in sync mode.

optional; bool; default False

HEDWIG_USE_TRANSPORT_MESSAGE_ATTRIBUTES

Flag indicating if meta attributes should be sent as transport message attributes. If set to False, meta attributes are sent as part of the payload - this is the legacy method for publishing metadata and newer apps should not change this value.

optional; bool; default True.

2.3 API reference

`hedwig.consumer.`**`listen_for_messages`**(*num_messages=10*, *visibility_timeout_s=None*, *shutdown_event=None*)

Starts a Hedwig listener for message types provided and calls the callback handlers like so:

```
callback_fn(message)
```

The message is explicitly deleted only if callback function ran successfully. In case of an exception the message is kept on queue and processed again. If the callback keeps failing, the message is moved to the dead-letter queue.

This function is blocking by default. It may be stopped by passing a shut down event object which can be set to stop the function.

Parameters

- **num_messages** (`int`) – Maximum number of messages to fetch in one API call. Defaults to 10
- **visibility_timeout_s** (`Optional[int]`) – The number of seconds the message should remain invisible to other queue readers. Defaults to None, which is queue default
- **shutdown_event** (`Optional[Event]`) – An event to signal that the process should shut down. This prevents more messages from being de-queued and function exits after the current messages have been processed.

Return type `None`

hedwig.consumer.**process_messages_for_lambda_consumer**(*lambda_event*)

Return type None

hedwig.conf.**settings**
This object allows settings to be accessed as properties. Settings can be configured in one of three ways:

1. Environment variable named SETTINGS_MODULE that points to a python module with settings as module attributes
2. Django - if Django can be imported, Django settings will be used automatically
3. Using an object or dict, by calling *hedwig.conf.settings.configure_with_object()*

Some setting values need to be string import paths will be automatically resolved and return the class.

Once settings have been configured, they shouldn't be changed. It is possible to re-configure for testing, but its not guaranteed to work the same way for non-test use cases.

hedwig.conf.settings.**configure_with_object**(*obj*)
Set Hedwig config using a dataclass-like object that contains all settings as its attributes, or a dict that contains settings as its keys.

Return type None

hedwig.conf.settings.**clear_cache**()
Clear settings cache - useful for testing only

Return type None

hedwig.conf.settings.**configured**
Have Hedwig settings been configured?

class hedwig.models.**Message**(*data*, *type*, *version*, *id=<factory>*, *metadata=<factory>*)
Model for Hedwig messages. A Message object will always have known message schema and schema version even if the data _may_ not be valid.

data: Any
Message data

type: str
Message type. May be none if message is invalid

version: distutils.version.StrictVersion
StrictVersion object representing data schema version.

id: str
Message identifier

metadata: *hedwig.models.Metadata*
Message metadata

static deserialize(*payload*, *attributes*, *provider_metadata*)
Deserialize a message from on-the-wire payload :raise: hedwig.ValidationError if validation fails.

Return type *Message*

static deserialize_firehose(*line*)
Deserialize a message from a line of firehose file :raise: hedwig.ValidationError if validation fails.

Return type *Message*

static deserialize_containerized(*payload*)
Deserialize a message assuming containerized format regardless of configured settings :raise: hedwig.ValidationError if validation fails.

Return type *Message*

classmethod new(*msg_type*, *version*, *data*, *msg_id=None*, *headers=None*)
Creates Message object given type, data schema version and data. This is typically used by the publisher code.

Parameters

- **msg_type** (Union[str, Enum]) – message type (could be an enum, it's value will be used)
- **version** (StrictVersion) – StrictVersion representing data schema
- **data** (Any) – The dict to pass in *data* field of Message.
- **msg_id** (Optional[str]) – Custom message identifier. If not passed, a randomly generated uuid will be used.
- **headers** (Optional[Dict[str, str]]) – Custom headers (keys must not begin with reserved namespace *hedwig_*)

Return type *Message*

publish()
Publish this message on Hedwig infra :rtype: Union[str, Future] :returns: for async publishers, returns a future that represents the publish api call, otherwise, returns the published message id

extend_visibility_timeout(*visibility_timeout_s*)
Extends visibility timeout of a message for long running tasks.

Return type None

property timestamp: int

Return type int

property headers: Dict[str, str]

Return type Dict[str, str]

property provider_metadata

property publisher: str

Return type str

serialize()
Serialize a message for appropriate on-the-wire format :rtype: Tuple[Union[str, bytes], dict] :return: Tuple of message payload and transport attributes

serialize_containerized()
Serialize a message using containerized format regardless of configured settings. In most cases, you just want to use *.serialize*. :rtype: Union[str, bytes] :return: Message payload

serialize_firehose()
Serialize a message for appropriate firehose file format. See hedwig.validators.base.HedwigBaseValidator.deserialize_firehose() for details. :rtype: str :return: Tuple of message payload and transport attributes

class hedwig.models.**Metadata**(*timestamp=<factory>*, *publisher=<factory>*, *headers=<factory>*, *provider_metadata=None*)

timestamp: int
Timestamp of message creation in epoch milliseconds

publisher: str
Publisher of message

headers: Dict[str, str]
Custom headers sent with the message

provider_metadata: Any = None
Provider specific metadata, such as SQS Receipt, or Google ack id. This may be used to extend message visibility if the task is running longer than expected using *Message.extend_visibility_timeout()*

class hedwig.validators.jsonschema.**JSONSchemaValidator**(*schema=None*)

checker = <FormatChecker checkers=['date', 'email', 'hostname', 'idn-email', 'idn-hostname', 'ipv4', 'ipv6', 'json-pointer', 'regex', 'relative-json-pointer', 'time']>
FormatChecker that checks for *format* JSON-schema field. This may be customized by an app by overriding setting *HEDWIG_DATA_VALIDATOR_CLASS* and defining more format checkers.

deserialize(*message_payload*, *attributes*, *provider_metadata*)
Deserialize a message from the on-the-wire format :type message_payload: Union[str, bytes] :param message_payload: Raw message payload as received from the backend :type provider_metadata: Any :param provider_metadata: Provider specific metadata :type attributes: dict :param attributes: Message attributes from the transport backend :rtype: *Message* :returns: Message object if valid :raise: hedwig.ValidationError if validation fails.

deserialize_containerized(*message_payload*)
Deserialize a message assuming containerized format regardless of configured setting. :type message_payload: Union[str, bytes] :param message_payload: Raw message payload as received from the backend :rtype: *Message* :returns: Message object if valid :raise: hedwig.ValidationError if validation fails.

deserialize_firehose(*line*)
Deserialize a message from a line in firehose file. This is slightly different from on-the-wire format:

1. It always uses container format (i.e. HEDWIG_USE_TRANSPORT_MESSAGE_ATTRIBUTES is ignored)
2. For binary file formats, its possible data is encoded as binary base64 blob rather than JSON.
3. Known minor versions isn't verified - knowing major version schema is good enough to read firehose.

Parameters **line** (str) – Raw line read from firehose file

Return type *Message*

Returns Message object if valid

Raise hedwig.ValidationError if validation fails.

serialize(*message*)
Serialize a message for appropriate on-the-wire format :rtype: Tuple[Union[str, bytes], dict] :return: Tuple of message payload and transport attributes

serialize_containerized(*message*)
Serialize a message using containerized format regardless of configured settings. In most cases, you just want to use *.serialize*. :rtype: Union[str, bytes] :return: Message payload

serialize_firehose(*message*)
Serialize a message for appropriate firehose file format. See hedwig.validators.base.HedwigBaseValidator.deserialize_firehose() for details. :rtype: str :return: Tuple of message payload and transport attributes

class hedwig.validators.protobuf.**ProtobufValidator**(*schema_module=None*)
A validator that encodes the payload using Protobuf binary format.

deserialize(*message_payload*, *attributes*, *provider_metadata*)
Deserialize a message from the on-the-wire format :type message_payload: Union[str, bytes] :param message_payload: Raw message payload as received from the backend :type provider_metadata: Any :param provider_metadata: Provider specific metadata :type attributes: dict :param attributes: Message attributes from the transport backend :rtype: *Message* :returns: Message object if valid :raise: hedwig.ValidationError if validation fails.

deserialize_containerized(*message_payload*)
Deserialize a message assuming containerized format regardless of configured setting. :type message_payload: Union[str, bytes] :param message_payload: Raw message payload as received from the backend :rtype: *Message* :returns: Message object if valid :raise: hedwig.ValidationError if validation fails.

deserialize_firehose(*line*)
Deserialize a message from a line in firehose file. This is slightly different from on-the-wire format:

1. It always uses container format (i.e. HEDWIG_USE_TRANSPORT_MESSAGE_ATTRIBUTES is ignored)
2. For binary file formats, its possible data is encoded as binary base64 blob rather than JSON.
3. Known minor versions isn't verified - knowing major version schema is good enough to read firehose.

Parameters **line** (str) – Raw line read from firehose file

Return type *Message*

Returns Message object if valid

Raise hedwig.ValidationError if validation fails.

serialize(*message*)
Serialize a message for appropriate on-the-wire format :rtype: Tuple[Union[str, bytes], dict] :return: Tuple of message payload and transport attributes

serialize_containerized(*message*)
Serialize a message using containerized format regardless of configured settings. In most cases, you just want to use *.serialize*. :rtype: Union[str, bytes] :return: Message payload

serialize_firehose(*message*)
Serialize a message for appropriate firehose file format. See hedwig.validators.base.HedwigBaseValidator.deserialize_firehose() for details. :rtype: str :return: Tuple of message payload and transport attributes

hedwig.commands.**requeue_dead_letter**(*num_messages=10*, *visibility_timeout=None*)
Re-queues everything in the Hedwig DLQ back into the Hedwig queue.

Parameters

- **num_messages** (int) – Maximum number of messages to fetch in one call. Defaults to 10.
- **visibility_timeout** (Optional[int]) – The number of seconds the message should remain invisible to other queue readers. Defaults to None, which is queue default

Return type None

class hedwig.backends.gcp.**GoogleMetadata**(*ack_id*, *subscription_path*, *publish_time*, *delivery_attempt*)
Google Pub/Sub specific metadata for a Message

ack_id: str
The ID used to ack the message

subscription_path: str
Path of the Pub/Sub subscription from which this message was pulled

publish_time: datetime.datetime
Time this message was originally published to Pub/Sub

delivery_attempt: int
The delivery attempt counter received from Pub/Sub. The first delivery of a given message will have this value as 1. The value is calculated as best effort and is approximate.

class hedwig.backends.aws.**AWSMetadata**(*receipt*, *first_receive_time*, *sent_time*, *receive_count*)
AWS specific metadata for a Message

receipt: str
AWS receipt identifier

2.3.1 Testing

hedwig.testing.pytest_plugin.**mock_hedwig_publish**()
A pytest fixture that mocks Hedwig publisher and lets you verify that your test publishes appropriate messages.

Return type Generator[*HedwigPublishMock*, None, None]

class hedwig.testing.pytest_plugin.**HedwigPublishMock**(**args*, ***kw*)
Custom mock class used by *hedwig.testing.pytest_plugin.mock_hedwig_publish()* to mock the publisher.

assert_message_not_published(*msg_type*, *data=None*, *version=StrictVersion('1.0')*)
Helper function to check that a Hedwig message of given type, data and schema was NOT sent.

Return type None

assert_message_published(*msg_type*, *data=None*, *version=StrictVersion('1.0')*)
Helper function to check if a Hedwig message with given type, data and schema version was sent.

Return type None

hedwig.testing.config.**unconfigure**()
If settings were configured, un-configure them - useful for testing only.

Return type None

2.3.2 Exceptions

class hedwig.exceptions.**RetryException**(**args*, ***kwargs*)
Special exception that does not log an exception when it is received. This is a retryable error.

class hedwig.exceptions.**IgnoreException**
Indicates that this task should be ignored.

class hedwig.exceptions.**ValidationError**
Message failed JSON schema validation

class hedwig.exceptions.**ConfigurationError**
There was some problem with settings

class hedwig.exceptions.**CallbackNotFound**
No callback found that can handle the given message. Check your *CALLBACKS* settings.

2.4 Release Notes

Current version: v8.9.1-dev

2.4.1 v1.0

- Initial version

2.4.2 v2.0

- Added support for Google Cloud Platform PubSub Service

2.5 Hedwig Migration Guide

2.5.1 v7 -> v8

The library now supports additional validators / serialization formats. Support for Protobuf has been added as a first class feature. To use Protobuf, set:

```
HEDWIG_DATA_VALIDATOR_CLASS = 'hedwig.validators.protobuf.ProtobufValidator'
HEDWIG_PROTOBUF_SCHEMA_MODULE = <fully qualified module path for compiled protos>
```

See protobuf usage and examples for additional instructions.

In addition, support for transport message attributes is now stable and is now the default. To continue using the "containerized" payload, set `HEDWIG_USE_TRANSPORT_MESSAGE_ATTRIBUTES = False`.

2.5.2 v6 -> v7

GCP Pub/Sub now natively supports dead-letter queues. As a result, support for managing retry state has been removed. If you used `MessageRetryStateBackend` or `MessageRetryStateRedis`, then ensure that the infrastructure is updated to set up dead-letter queues in GCP. hedwig-subscription Terraform module v3 adds support for DLQ.

2.5.3 v5 -> v6

GCP infrastructure doesn't assume Dataflow any more since that requires a potentially expensive component. Instead, the library supports subscribing to multiple subscriptions from an app. hedwig-subscription Terraform module v2 removes Dataflow infra. Update settings:

```
HEDWIG_SUBSCRIPTIONS = <LIST OF YOUR HEDWIG TOPIC NAMES APP IS SUBSCRIBED TO>
```

2.5.4 v4 -> v5

`hedwig.models.MessageType` enum has been removed. Instead the library supports message types being passed as either Enum value, or strings. The use of Enum classes is left up to the application.

2.5.5 v3 → v4

The library now supports additional transport backends. Support for Google Pub/Sub has been added as a first class feature. To use Google Pub/Sub, set:

```
HEDWIG_CONSUMER_BACKEND = 'hedwig.backends.gcp.GooglePubSubConsumerBackend'
HEDWIG_PUBLISHER_BACKEND = 'hedwig.backends.gcp.GooglePubSubPublisherBackend'
```

To continue using AWS, set:

```
HEDWIG_PUBLISHER_BACKEND = 'hedwig.backends.aws.AWSSNSPublisherBackend'
HEDWIG_CONSUMER_BACKEND = 'hedwig.backends.aws.AWSSQSConsumerBackend'
```

See configuration quickstart for additional instructions.

2.5.6 v2 -> v3

There are no imports at top-level available any more. Change all imports from `hedwig.` to specific modules.

2.5.7 v1 -> v2

Update schema file and change instances of `<major>.<minor>` to `<major>.*` and remove all non-latest minor versions for every message type. Just one major version schema should suffice since there should only be non-breaking changes.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX