Happyly Documentation

Release 0.7.0alpha1

Alexander Tsukanov

Apr 09, 2019

CONTENTS:

CHAPTER ONE

INTRODUCTION

Happyly is a scalable solution for systems which handle any kind of messages. Have you ever seen a codebase where serialization, acknowledgement and business logic are mixed together like a spaghetti? I have. Imagine switching between Google Pub/Sub and Django REST Framework. Or Celery. This shouldn't be a nightmare but it often is.

Here's the approach of Happyly:

- Write you business logic in universal *Handlers*, which don't care at all how you serialize things or send them over network or deal with message queues.
- Describe your schemas using ORM/Framework-agnostic technology.
- Plug-in any details of messaging protocol, serialization and networking. Change them with different drop-in replacements at any time.

1.1 Use cases

- **Google Pub/Sub**

  Let's be honest, the official Python client library is too low-level. You must serialize and deserialize things manually, as well as to `ack` and `nack` messages.

  Usual way:

```
def callback(message):
    attributes = json.loads(message.data)
    try:
        result = process_things(attributes['ID'])
        encoded = json.dumps(result).encode('utf-8')
        PUBLISHER.publish(TOPIC, encoded)
    except NeedToRetry:
        _LOGGER.info('Not acknowledging, will retry later.')
    except Exception:
        _LOGGER.error('An error occured')
        message.ack()
    else:
        message.ack()
```

  Happyly way:

```
class MyHandler(happyly.handler):
    def handle(attributes: dict):
        return process_things(attributes['ID'])
```

(continues on next page)

(continued from previous page)

```
    def on_handling_failed(attributes: dict, error):
        if isinstance(error, NeedToRetry):
            raise error from error
        else:
            _LOGGER.error('An error occured')
```

`MyHandler` is now also usable with Celery or Flask. Or with yaml serialization. Or with `message.attributes` instead of `message.data`. Without any change.

- You are going to **change messaging technology** later.

  Easy! Here's an example.

  1. Define your message schemas.

```
class MyInputSchema(happyly.Schema):
    request_id = marshmallow.fields.Str(required=True)

class MyOutputSchema(happyly.Schema):
    request_id = marshmallow.fields.Str(required=True)
    result = marshmallow.fields.Str(required=True)
    error = marshmallow.fields.Str()
```

  2. Define your handler

```
class ProcessThings(happyly.Handler):
    def handle(message: dict):
        req_id = message['request_id']
        if req_id in ALLOWED:
            result = get_result_for_id(req_id)
        else:
            result = 'not allowed'
        return {
            'request_id': req_id
            'result': result
        }

    def on_handling_failed(message: dict, error):
        return {
            'request_id': message['request_id']
            'result': 'error',
            'error': str(error)
        }
```

  3. Plug it into Celery:

```
@celery.task('hello')
def hello(message):
    result = happyly.Executor(
        handler=ProcessThings(),
        serializer=happyly.DummyValidator(schema=MyInputSchema()),
        deserializer=happyly.DummyValidator(schema=MyOutputSchema()),
    ).run_for_result(
        message
    )
    return result
```

  4. Or Google Pub/Sub:

```
happyly.Listener(
    handler=ProcessThings(),
    deserializer=happyly.google_pubsub.JSONDeserializerWithRequestIdRequired(
        schema=MyInputSchema()
    ),
    serializer=happyly.google_pubsub.BinaryJSONSerializer(
        schema=MyOutputSchema()
    ),
    publisher=happyly.google_pubsub.GooglePubSubPublisher(
        topic='my_topic',
        project='my_project',
    ),
 ).start_listening()
```

5. Move to any other technology. Or swap serializer to another. Do whatever you need while your handler and schemas remain absolutely the same.

CHAPTER TWO

INSTALLATION

Happyly is hosted on PyPI, so you can use:

```
pip install happyly
```

There is an extra dependency which enables cached components via Redis. If you need it, install it like this:

```
pip install happyly[redis]
```

CHAPTER

THREE

CONCEPTS

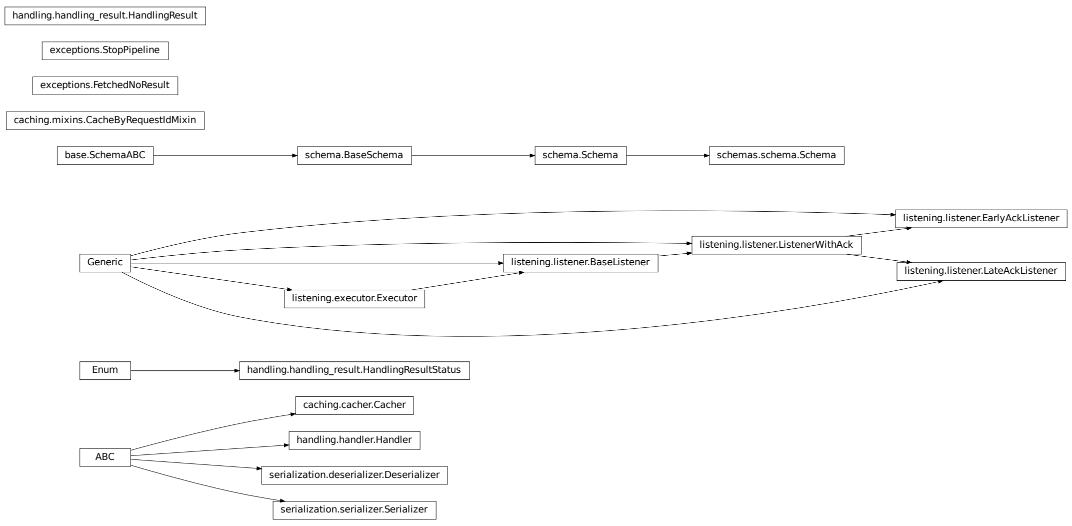
CHAPTER

FOUR

ADVANCED

CHAPTER FIVE

API REFERENCE


handling.handling_result.HandlingResult
exceptions.StopPipeline
exceptions.FetchedNoResult
caching.mixins.CacheByRequestIdMixin
base.SchemaABC
schema.BaseSchema
schema.Schema
schemas.schema.Schema
Generic
listening.listener.EarlyAckListener
listening.listener.ListenerWithAck
listening.listener.BaseListener
listening.listener.LateAckListener
listening.executor.Executor
Enum
handling.handling_result.HandlingResultStatus
caching.cacher.Cacher
ABC
handling.handler.Handler
serialization.deserializer.Deserializer
serialization.serializer.Serializer

happyly.listening.executor	
happyly.listening.listener	*BaseListener* and its subclasses.
happyly.schemas.schema	
happyly.caching.cacher	
happyly.caching.mixins	
happyly.serialization.serializer	
happyly.serialization.deserializer	
happyly.handling.handler	
happyly.handling.handling_result	
happyly.handling.dummy_handler._DummyHandler	
happyly.exceptions	

5.1 happyly.listening.executor

Description

Classes

Executor([handler, deserializer, publisher, . . .])	Component which is able to run handler as a part of more complex pipeline.
ResultAndDeserialized(result, deserialized)	Create new instance of ResultAndDeserialized(result, deserialized)

5.1.1 happyly.listening.executor.Executor

class happyly.listening.executor.**Executor**(*handler=<happyly.handling.dummy_handler._DummyHandler object>, deserializer=<happyly.serialization.dummy.DummySerde object>, publisher=None, serializer=<happyly.serialization.dummy.DummySerde object>*)

Bases: `typing.Generic`

Component which is able to run handler as a part of more complex pipeline.

Implements managing of stages inside the pipeline (deserialization, handling, serialization, publishing) and introduces callbacks between the stages which can be easily overridden.

Executor does not implement stages themselves, it takes internal implementation of stages from corresponding components: `Handler`, `Deserializer`, `Publisher`.

It means that *Executor* is universal and can work with any serialization/messaging technology depending on concrete components provided to executor's constructor.

deserializer	Provides implementation of deserialization stage to Executor.
handler	Provides implementation of handling stage to Executor.
publisher	Provides implementation of serialization and publishing stages to Executor.
serializer	
on_deserialization_failed(message, error)	Callback which is called right after deserialization failure.
on_deserialized(original_message, parsed_message)	Callback which is called right after message was deserialized successfully.
on_finished(original_message, error)	Callback which is called when pipeline finishes its execution.
on_handled(original_message, parsed_message, . . .)	Callback which is called right after message was handled (successfully or not, but without raising an exception).
on_handling_failed(original_message, . . .)	Callback which is called if handler's `on_handling_failed` raises an exception.
on_published(original_message, . . .)	Callback which is called right after message was published successfully.
on_publishing_failed(original_message, . . .)	Callback which is called when publisher fails to publish.
on_received(message)	Callback which is called as soon as pipeline is run.

Continued on next page

Table 3 – continued from previous page

`on_serialization_failed`(original, . . .)	
`on_serialized`(original, deserialized, . . .)	
`on_stopped`(original_message[, reason])	Callback which is called when pipeline is stopped via *`StopPipeline`*
`run`([message])	Method that starts execution of pipeline stages.
`run_for_result`([message])	

`handler = <happyly.handling.dummy_handler._DummyHandler object>`

Provides implementation of handling stage to Executor.

Type: *`Handler`*

`deserializer = <happyly.serialization.dummy.DummySerde object>`

Provides implementation of deserialization stage to Executor.

If not present, no deserialization is performed.

Type: ~D

`publisher = None`

Provides implementation of serialization and publishing stages to Executor.

If not present, no publishing is performed.

Type: `Optional`[~P]

`on_received`(*message*)

Callback which is called as soon as pipeline is run.

Override it in your custom Executor/Listener if needed, but don't forget to call implementation from base class.

Parameters **`message`** (`Any`) – Message as it has been received, without any deserialization

`on_deserialized`(*original_message*, *parsed_message*)

Callback which is called right after message was deserialized successfully.

Override it in your custom Executor/Listener if needed, but don't forget to call implementation from base class.

Parameters

- **`original_message`** (`Any`) – Message as it has been received, without any deserialization
- **`parsed_message`** (`Mapping[str, Any]`) – Message attributes after deserialization

`on_deserialization_failed`(*message*, *error*)

Callback which is called right after deserialization failure.

Override it in your custom Executor/Listener if needed, but don't forget to call implementation from base class.

Parameters

- **`message`** (`Any`) – Message as it has been received, without any deserialization
- **`error`** (`Exception`) – exception object which was raised

`on_handled`(*original_message*, *parsed_message*, *result*)

Callback which is called right after message was handled (successfully or not, but without raising an exception).

Override it in your custom Executor/Listener if needed, but don't forget to call implementation from base class.

Parameters

- **`original_message`** (`Any`) – Message as it has been received, without any deserialization
- **`parsed_message`** (`Mapping[str, Any]`) – Message attributes after deserialization
- **`result`** (*`HandlingResult`*) – Result fetched from handler (also shows if handling was successful)

`on_handling_failed`(*original_message*, *parsed_message*, *error*)

Callback which is called if handler's `on_handling_failed` raises an exception.

Override it in your custom Executor/Listener if needed, but don't forget to call implementation from base class.

Parameters

- **`original_message`** (`Any`) – Message as it has been received, without any deserialization
- **`parsed_message`** (`Mapping[str, Any]`) – Message attributes after deserialization
- **`error`** (`Exception`) – exception object which was raised

`on_published`(*original_message*, *parsed_message*, *result*)

Callback which is called right after message was published successfully.

Override it in your custom Executor/Listener if needed, but don't forget to call implementation from base class.

Parameters

- **`original_message`** (`Any`) – Message as it has been received, without any deserialization
- **`parsed_message`** (`Optional[Mapping[str, Any]]`) – Message attributes after deserialization
- **`result`** (*`HandlingResult`*) – Result fetched from handler (also shows if handling was successful)

`on_publishing_failed`(*original_message*, *parsed_message*, *result*, *error*)

Callback which is called when publisher fails to publish.

Override it in your custom Executor/Listener if needed, but don't forget to call implementation from base class.

Parameters

- **`original_message`** (`Any`) – Message as it has been received, without any deserialization
- **`parsed_message`** (`Optional[Mapping[str, Any]]`) – Message attributes after deserialization
- **`result`** (*`HandlingResult`*) – Result fetched from handler (also shows if handling was successful)
- **`error`** (`Exception`) – exception object which was raised

on_finished(*original_message*, *error*)
Callback which is called when pipeline finishes its execution. Is guaranteed to be called unless pipeline is stopped via StopPipeline.

Parameters

- **original_message** (Any) – Message as it has been received, without any deserialization
- **error** (Optional[Exception]) – exception object which was raised or None

on_stopped(*original_message*, *reason=''*)
Callback which is called when pipeline is stopped via *StopPipeline*

Parameters

- **original_message** (Any) – Message as it has been received, without any deserialization
- **reason** (str) – message describing why the pipeline stopped

run(*message=None*)
Method that starts execution of pipeline stages.

To stop the pipeline raise StopPipeline inside any callback.

Parameters message (Optional[Any]) – Message as is, without deserialization. Or message attributes if the executor was instantiated with neither a deserializer nor a handler (useful to quickly publish message attributes by hand)

5.1.2 happyly.listening.executor.ResultAndDeserialized

class happyly.listening.executor.**ResultAndDeserialized**(*result*, *deserialized*)
Bases: tuple

Create new instance of ResultAndDeserialized(result, deserialized)

deserialized	Alias for field number 1
result	Alias for field number 0

deserialized
Alias for field number 1

result
Alias for field number 0

5.2 happyly.listening.listener

Description

BaseListener and its subclasses. Listener is a form of Executor which is able to run pipeline by an event coming from a subscription.

Classes

`BaseListener`(subscriber, handler, deserializer)	Listener is a form of Executor which is able to run pipeline by an event coming from a subscription.
`EarlyAckListener`(subscriber, handler, . . . [, . . .])	Acknowledge-aware `BaseListener`, which performs `ack()` right after `on_received()` callback is finished.
`LateAckListener`(subscriber, handler, . . . [, . . .])	Acknowledge-aware listener, which performs `ack()` at the very end of pipeline.
`ListenerWithAck`(subscriber, handler, . . . [, . . .])	Acknowledge-aware listener.

5.2.1 happyly.listening.listener.BaseListener

class `happyly.listening.listener.`**BaseListener**(*subscriber*, *handler*, *deserializer*, *serializer=<happyly.serialization.dummy.DummySerde object>*, *publisher=None*)

Bases: `happyly.listening.executor.Executor`, `typing.Generic`

Listener is a form of Executor which is able to run pipeline by an event coming from a subscription.

Listener itself doesn't know how to subscribe, it subscribes via a provided subscriber.

As any executor, implements managing of stages inside the pipeline (deserialization, handling, serialization, publishing) and contains callbacks between the stages which can be easily overridden.

As any executor, listener does not implement stages themselves, it takes internal implementation of stages from corresponding components: handler, deserializer, publisher.

It means that listener is universal and can work with any serialization/messaging technology depending on concrete components provided to listener's constructor.

`start_listening()`

subscriber = None

Provides implementation of how to subscribe.

Type: ~S

5.2.2 happyly.listening.listener.EarlyAckListener

class `happyly.listening.listener.`**EarlyAckListener**(*subscriber*, *handler*, *deserializer*, *serializer=<happyly.serialization.dummy.DummySerde object>*, *publisher=None*)

Bases: `happyly.listening.listener.ListenerWithAck`, `typing.Generic`

Acknowledge-aware `BaseListener`, which performs `ack()` right after `on_received()` callback is finished.

5.2.3 happyly.listening.listener.LateAckListener

class happyly.listening.listener.**LateAckListener**(*subscriber*, *handler*, *deserializer*, *serializer=<happyly.serialization.dummy.DummySerde object>*, *publisher=None*)

Bases: *happyly.listening.listener.ListenerWithAck*, typing.Generic

Acknowledge-aware listener, which performs *ack()* at the very end of pipeline.

on_finished(original_message, error)	Callback which is called when pipeline finishes its execution.

on_finished(*original_message*, *error*)

Callback which is called when pipeline finishes its execution. Is guaranteed to be called unless pipeline is stopped via StopPipeline.

Parameters

- **original_message** (Any) – Message as it has been received, without any deserialization
- **error** (Optional[Exception]) – exception object which was raised or None

5.2.4 happyly.listening.listener.ListenerWithAck

class happyly.listening.listener.**ListenerWithAck**(*subscriber*, *handler*, *deserializer*, *serializer=<happyly.serialization.dummy.DummySerde object>*, *publisher=None*)

Bases: *happyly.listening.listener.BaseListener*, typing.Generic

Acknowledge-aware listener. Defines *ListenerWithAck.ack()* method. Subclass *ListenerWithAck* and specify when to ack by overriding the corresponding callbacks.

ack(message)	Acknowledge the message using implementation from subscriber, then log success.
on_acknowledged(message)	Callback which is called write after message was acknowledged.

on_acknowledged(*message*)

Callback which is called write after message was acknowledged.

Override it in your custom Executor/Listener if needed, but don't forget to call implementation from base class.

Parameters message (Any) – Message as it has been received, without any deserialization

ack(*message*)

Acknowledge the message using implementation from subscriber, then log success.

Parameters message (Any) – Message as it has been received, without any deserialization

5.3 happyly.schemas.schema

Description

Classes

Schema(*args, **kwargs)	Marshmallow schema, which raises errors on mismatch (extra fields provided also raise exception).

5.3.1 happyly.schemas.schema.Schema

class `happyly.schemas.schema.`**`Schema`**(**args*, ***kwargs*)

Bases: `marshmallow.schema.Schema`

Marshmallow schema, which raises errors on mismatch (extra fields provided also raise exception).

Subclass it just like any marshmallow `Schema` to describe schema.

Instantiation with no arguments is a good strict default, but you can pass any arguments valid for `marshmallow.Schema`

`opts`
`check_unknown_fields`(data, original_data)

5.4 happyly.caching.cacher

Description

Classes

Cacher	Abstract base class which defines interface of any caching component to be used via *CacheByRequestIdMixin* or similar mixin.

5.4.1 happyly.caching.cacher.Cacher

class `happyly.caching.cacher.`**`Cacher`**

Bases: `abc.ABC`

Abstract base class which defines interface of any caching component to be used via *CacheByRequestIdMixin* or similar mixin.

add(data, key)	Add the provided data to cache and store it by the provided key.
get(key)	Returns data which is stored in cache by the provided key.
remove(key)	Remove data from cache which is stored by the provided key.

add(*data*, *key*)

Add the provided data to cache and store it by the provided key.

remove(*key*)

Remove data from cache which is stored by the provided key.

get (*key*)
Returns data which is stored in cache by the provided key.

5.5 happyly.caching.mixins

Description

Classes

CacheByRequestIdMixin(cacher)	Mixin which adds caching functionality to Listener.

5.5.1 happyly.caching.mixins.CacheByRequestIdMixin

class happyly.caching.mixins.**CacheByRequestIdMixin** (*cacher*)
Bases: `object`

Mixin which adds caching functionality to Listener. Utilizes notions of listener's topic and request id of message – otherwise will not work.

To be used via multiple inheritance. For example, given some component `SomeListener` you can define its caching equivalent by defining `SomeCachedListener` which inherits from both `SomeListener` and *CacheByRequestIdMixin*.

`on_deserialization_failed`(message, error)	
`on_published`(original_message, …)	
`on_received`(message)	

5.6 happyly.serialization.serializer

Description

Classes

Serializer	Abstract base class for Serializer.

5.6.1 happyly.serialization.serializer.Serializer

class happyly.serialization.serializer.**Serializer**
Bases: `abc.ABC`

Abstract base class for Serializer. Provides `serialize()` method which should be implemented by subclasses.

`serialize`(message_attributes)

rtype `Any`

5.7 happyly.serialization.deserializer

Description

Classes

`Deserializer`	

5.7.1 happyly.serialization.deserializer.Deserializer

`class` `happyly.serialization.deserializer.`**`Deserializer`**

Bases: `abc.ABC`

`build_error_result`(message, error)

rtype `Mapping[str, Any]`

`deserialize`(message)

rtype `Mapping[str, Any]`

5.8 happyly.handling.handler

Description

Classes

`Handler`	A class containing logic to handle a parsed message.

5.8.1 happyly.handling.handler.Handler

`class` `happyly.handling.handler.`**`Handler`**

Bases: `abc.ABC`

A class containing logic to handle a parsed message.

`handle`(message)	Applies logic using a provided message, optionally gives back one or more results.
`on_handling_failed`(message, error)	Applies fallback logic using a provided message when *`handle()`* fails, optionally gives back one or more results.

`handle`(*message*)

Applies logic using a provided message, optionally gives back one or more results. Each result consists of message attributes which can be serialized and sent. When fails, calls *on_handling_failed()*

Parameters **message** (Mapping[str, Any]) – A parsed message as a dictionary of attributes

Return type Union[Mapping[str, Any], List[Mapping[str, Any]], None]

Returns None if no result is extracted from handling, a dictionary of attributes for single result or a list of dictionaries if handling provides multiple results

on_handling_failed(*message*, *error*)

Applies fallback logic using a provided message when *handle()* fails, optionally gives back one or more results. Enforces users of *Handler* class to provide explicit strategy for errors.

If you want to propagate error further to the underlying Executor/Handler, just re-raise an `error` here:

```
def on_handling_failed(self, message, error):
    raise error
```

Parameters

- **message** (Mapping[str, Any]) – A parsed message as a dictionary of attributes
- **error** (Exception) – Error raised by *handle()*

Return type Union[Mapping[str, Any], List[Mapping[str, Any]], None]

Returns None if no result is extracted from handling, a dictionary of attributes for single result or a list of dictionaries if handling provides multiple results

5.9 happyly.handling.handling_result

Description

Classes

HandlingResult(status, data)	
HandlingResultStatus	Handling status: `OK` or `ERR`

5.9.1 happyly.handling.handling_result.HandlingResult

class happyly.handling.handling_result.**HandlingResult**(*status*, *data*)

Bases: `object`

data	Result or results of handling.
status	Status of message handling.
err(data)	Construct failed *HandlingResult*.
ok(data)	Construct successful *HandlingResult*.

status = NOTHING

Status of message handling.

Type: *HandlingResultStatus*

data = NOTHING

Result or results of handling.

Type: Union[Mapping[str, Any], List[Mapping[str, Any]], None]

classmethod ok(*data*)

Construct successful *HandlingResult*.

Parameters data (Union[Mapping[str, Any], List[Mapping[str, Any]], None]) – message or list of messages which were processed.

Return type *HandlingResult*

classmethod err(*data*)

Construct failed *HandlingResult*.

Parameters data (Union[Mapping[str, Any], List[Mapping[str, Any]], None]) – message or list of messages which were processed.

Return type *HandlingResult*

5.9.2 happyly.handling.handling_result.HandlingResultStatus

class happyly.handling.handling_result.**HandlingResultStatus**

Bases: enum.Enum

Handling status: OK or ERR

ERR
OK

5.10 happyly.handling.dummy_handler._DummyHandler

class happyly.handling.dummy_handler.**_DummyHandler**

Bases: *happyly.handling.handler.Handler*

handle(message)	Applies logic using a provided message, optionally gives back one or more results.
on_handling_failed(message, error)	Applies fallback logic using a provided message when *handle()* fails, optionally gives back one or more results.

handle(*message*)

Applies logic using a provided message, optionally gives back one or more results. Each result consists of message attributes which can be serialized and sent. When fails, calls *on_handling_failed()*

Parameters message (Mapping[str, Any]) – A parsed message as a dictionary of attributes

Returns None if no result is extracted from handling, a dictionary of attributes for single result or a list of dictionaries if handling provides multiple results

on_handling_failed(*message*, *error*)

Applies fallback logic using a provided message when *handle()* fails, optionally gives back one or more results. Enforces users of Handler class to provide explicit strategy for errors.

If you want to propagate error further to the underlying Executor/Handler, just re-raise an error here:

```
def on_handling_failed(self, message, error):
    raise error
```

Parameters

- **message** (Mapping[str, Any]) – A parsed message as a dictionary of attributes
- **error** (Exception) – Error raised by *handle()*

Returns None if no result is extracted from handling, a dictionary of attributes for single result or a list of dictionaries if handling provides multiple results

5.11 happyly.exceptions

Description

Exceptions

FetchedNoResult()	
StopPipeline([reason])	This exception should be raised to stop a pipeline.

5.11.1 happyly.exceptions.FetchedNoResult

exception happyly.exceptions.**FetchedNoResult**

5.11.2 happyly.exceptions.StopPipeline

exception happyly.exceptions.**StopPipeline**(*reason=''*)

This exception should be raised to stop a pipeline. After raising it, `Executor.on_stopped()` will be called.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX