
Happyly Documentation

Release 0.8.0rc4

Alexander Tsukanov

May 02, 2019

CONTENTS:

1	Introduction	1
1.1	Use cases	1
2	Installation	5
3	Concepts	7
3.1	Handler	7
3.2	Executor	8
4	Advanced	9
5	API Reference	11
5.1	happyly.listening.executor	11
5.1.1	happyly.listening.executor.Executor	11
5.1.2	happyly.listening.executor.ResultAndDeserialized	14
5.2	happyly.listening.listener	15
5.2.1	happyly.listening.listener.BaseListener	15
5.2.2	happyly.listening.listener.EarlyAckListener	16
5.2.3	happyly.listening.listener.LateAckListener	16
5.2.4	happyly.listening.listener.ListenerWithAck	16
5.3	happyly.schemas.schema	17
5.3.1	happyly.schemas.schema.Schema	17
5.4	happyly.caching.cacher	17
5.4.1	happyly.caching.cacher.Cacher	17
5.5	happyly.caching.mixins	18
5.5.1	happyly.caching.mixins.CacheByRequestIdMixin	18
5.6	happyly.serialization.serializer	18
5.6.1	happyly.serialization.serializer.Serializer	19
5.6.2	happyly.serialization.serializer.SerializerWithSchema	19
5.7	happyly.serialization.deserializer	19
5.7.1	happyly.serialization.deserializer.Deserializer	19
5.7.2	happyly.serialization.deserializer.DeserializerWithSchema	20
5.8	happyly.handling.handler	20
5.8.1	happyly.handling.handler.Handler	20
5.9	happyly.handling.dummy_handler._DummyHandler	21
5.10	happyly.exceptions	21
5.10.1	happyly.exceptions.FetchedNoResult	22
5.10.2	happyly.exceptions.StopPipeline	22
6	Indices and tables	23

INTRODUCTION

Happyly is a scalable solution for systems which handle any kind of messages.

Happyly helps to abstract your business logic from messaging stuff, so that your code is maintainable and ensures separation of concerns.

Have you ever seen a codebase where serialization, message queue managing 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 etc.
- 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.

Happyly can be used with Flask, Celery, Django, Kafka or whatever technology which can be used for messaging and also provides first-class support of Google Pub/Sub.

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 occurred')
        message.ack()
    else:
        message.ack()
```

Happyly way:

```
def handle_my_stuff(message: dict):
    try:
        return process_things(message['ID'])
    except NeedToRetry as error:
        raise error from error
    except Exception:
        _LOGGER.error('An error occurred')
```

`handle_my_stuff` 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.

Let's say you are prototyping your project with Flask and are planning to move to Celery for better fault tolerance then. Or to Google Pub/Sub. You just haven't decided yet.

Easy! Here's how Happyly can help.

1. Define your message schemas.

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

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

2. Define your handler

```
def handle_things(message: dict):
    try:
        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
        }
    except Exception as error:
        return {
            'request_id': message['request_id'],
            'result': 'error',
            'error': str(error)
        }
```

3. Plug it into Flask:

```
@app.route('/', methods=['POST'])
def root():
    executor = happily.Executor(
        handler=handle_things,
        deserializer=DummyValidator(schema=MyInputSchema()),
        serializer=JsonifyForSchema(schema=MyOutputSchema()),
    )
```

(continues on next page)

(continued from previous page)

```
request_data = request.get_json()
return executor.run_for_result(request_data)
```

3. Painlessly switch to Celery when you need:

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

4. Or to Google Pub/Sub:

```
happily.Listener(
    handler=ProcessThings(),
    deserializer=happily.google_pubsub.JSONDeserializerWithRequestIdRequired(
        schema=MyInputSchema()
    ),
    serializer=happily.google_pubsub.BinaryJSONSerializer(
        schema=MyOutputSchema()
    ),
    publisher=happily.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.

INSTALLATION

Happyly is hosted on PyPI, so you can use:

```
pip install happily
```

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

```
pip install happily[redis]
```


CONCEPTS

3.1 Handler

Handler is the main concept of all Happyly library. Basically a handler is a callable which implements business logic, and nothing else:

- No serialization/deserialization here
- No sending stuff over the network
- No message queues' related stuff

Let the handler do its job!

To create a handler you can simply define a function which takes a `dict` as an input and returns a `dict`:

```
def handle_my_stuff(message: dict):
    try
        db.update(message['user'], message['status'])
        return {
            'request_id': message['request_id'],
            'action': 'updated',
        }
    except Exception:
        return {
            'action': 'failed'
        }
```

Done! This handler can be plugged into your application: whether it uses Flask or Celery or whatever.

Note that you are allowed to return nothing if you don't actually need a result from your handler. This handler is also valid:

```
def handle_another_stuff(message: dict):
    try
        neural_net.start_job(message['id'])
        _LOGGER.info('Job created')
    except Exception:
        _LOGGER.warning('Failed to create a job')
```

If you prefer class-based approach, Happyly can satisfy you too. Subclass `happyly.Handler()` and implement the following methods:

```
class MyHandler(happyly.Handler):
```

(continues on next page)

(continued from previous page)

```
def handle(message: dict)
    db.update(message['user'], message['status'])
    return {
        'request_id': message['request_id'],
        'action': 'updated',
    }

def on_handling_failed(message: dict, error)
    return {
        'action': 'failed'
    }
```

Instance of `MyHandler` is equivalent to `handle_my_stuff`

3.2 Executor

To plug a handler into your application you will need `happyly.Executor()` (or one of its subclasses).

ADVANCED

API REFERENCE

<code>happyly.listening.executor</code>	
<code>happyly.listening.listener</code>	<i>BaseListener</i> and its subclasses.
<code>happyly.schemas.schema</code>	
<code>happyly.caching.cacher</code>	
<code>happyly.caching.mixins</code>	
<code>happyly.serialization.serializer</code>	
<code>happyly.serialization.deserializer</code>	
<code>happyly.handling.handler</code>	
<code>happyly.handling.handling_result</code>	
<code>happyly.handling.dummy_handler._DummyHandler</code>	
<code>happyly.exceptions</code>	

5.1 happyly.listening.executor

Description

Classes

<code>Executor([handler, deserializer, publisher, ...])</code>	Component which is able to run handler as a part of more complex pipeline.
<code>ResultAndDeserialized(result, deserialized)</code>	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=None, publisher=None, serializer=None*)

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.

<code>on_deserialization_failed(original_message, ...)</code>	Callback which is called right after deserialization failure.
<code>on_deserialized(original_message, ...)</code>	Callback which is called right after message was deserialized successfully.
<code>on_finished(original_message, error)</code>	Callback which is called when pipeline finishes its execution.
<code>on_handled(original_message, ...)</code>	Callback which is called right after message was handled (successfully or not, but without raising an exception).
<code>on_handling_failed(original_message, ...)</code>	Callback which is called if handler's <code>on_handling_failed</code> raises an exception.
<code>on_published(original_message, ...)</code>	Callback which is called right after message was published successfully.
<code>on_publishing_failed(original_message, ...)</code>	Callback which is called when publisher fails to publish.
<code>on_received(original_message)</code>	Callback which is called as soon as pipeline is run.
<code>on_serialization_failed(original, ...)</code>	
<code>on_serialized(original_message, ...)</code>	
<code>on_stopped(original_message[, reason])</code>	Callback which is called when pipeline is stopped via <code>StopPipeline</code>
<code>run([message])</code>	Method that starts execution of pipeline stages.
<code>run_for_result([message])</code>	

handler = None

Provides implementation of handling stage to Executor.

Type: `Union[Handler, Callable[[Mapping[str, Any]], Optional[Mapping[str, Any]]]]`

deserializer = None

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 (original_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 `original_message` (`Any`) – Message as it has been received, without any deserialization

on_deserialized (original_message, deserialized_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
- **deserialized_message** (*Mapping[str, Any]*) – Message attributes after deserialization

on_deserialization_failed (*original_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

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

on_handled (*original_message, deserialized_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
- **deserialized_message** (*Mapping[str, Any]*) – Message attributes after deserialization
- **result** (*Optional[Mapping[str, Any]]*) – Result fetched from handler

on_handling_failed (*original_message, deserialized_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
- **deserialized_message** (*Mapping[str, Any]*) – Message attributes after deserialization
- **error** (*Exception*) – exception object which was raised

on_published (*original_message, deserialized_message, result, serialized_message*)

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

- **deserialized_message** (`Optional[Mapping[str, Any]]`) – Message attributes after deserialization
- **result** (`Optional[Mapping[str, Any]]`) – Result fetched from handler

on_publishing_failed (*original_message*, *deserialized_message*, *result*, *serialized_message*, *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
- **deserialized_message** (`Optional[Mapping[str, Any]]`) – Message attributes after deserialization
- **result** (`Optional[Mapping[str, Any]]`) – Result fetched from handler
- **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 happily.listening.executor.ResultAndDeserialized

class `happily.listening.executor.ResultAndDeserialized` (*result*, *deserialized*)

Bases: `tuple`

Create new instance of `ResultAndDeserialized(result, deserialized)`

deserialized

Alias for field number 1

Continued on next page

Table 4 – continued from previous page

<i>result</i>	Alias for field number 0
deserialized	Alias for field number 1
result	Alias for field number 0

5.2 happily.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

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

5.2.1 happily.listening.listener.BaseListener

class happily.listening.listener.**BaseListener**(*subscriber, handler, deserializer, serializer=<happily.serialization.dummy.DummySerializer>, publisher=None*)

Bases: *happily.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, serial-  
 izer=<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, serial-  
 izer=<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.

<code>on_finished(original_message, error)</code>	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, serial-  
 izer=<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.

<code>ack(message)</code>	Acknowledge the message using implementation from subscriber, then log success.
<code>on_acknowledged(message)</code>	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

<code>Schema(*args, **kwargs)</code>	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`

<code>opts</code>
<code>check_unknown_fields(data, original_data)</code>

5.4 happyly.caching.cacher

Description

Classes

<code>Cacher</code>	Abstract base class which defines interface of any caching component to be used via <code>CacheByRequestIdMixin</code> 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.

<i>add</i> (data, key)	Add the provided data to cache and store it by the provided key.
<i>get</i> (key)	Returns data which is stored in cache by the provided key.
<i>remove</i> (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 happily.caching.mixins

Description

Classes

<i>CacheByRequestIdMixin</i> (<i>cache</i>)	Mixin which adds caching functionality to Listener.
---	---

5.5.1 happily.caching.mixins.CacheByRequestIdMixin

class happily.caching.mixins.**CacheByRequestIdMixin** (*cache*)

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*.

<i>on_deserialization_failed</i> (<i>message</i> , <i>error</i>)
<i>on_published</i> (<i>original_message</i> , ...)
<i>on_received</i> (<i>message</i>)

5.6 happily.serialization.serializer

Description

Classes

<i>Serializer</i>	Abstract base class for Serializer.
<i>SerializerWithSchema</i> (schema)	

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.

<code>from_function(func)</code>	
<code>serialize(message_attributes)</code>	rtype <code>Any</code>

5.6.2 happyly.serialization.serializer.SerializerWithSchema

class happyly.serialization.serializer.**SerializerWithSchema** (*schema*)

Bases: `happyly.serialization.serializer.Serializer`, `abc.ABC`

<code>schema</code>

5.7 happyly.serialization.deserializer

Description

Classes

<i>Deserializer</i>
<i>DeserializerWithSchema</i> (schema)

5.7.1 happyly.serialization.deserializer.Deserializer

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

Bases: `abc.ABC`

<code>build_error_result(message, error)</code>	rtype <code>Mapping[str, Any]</code>
<code>deserialize(message)</code>	rtype <code>Mapping[str, Any]</code>
<code>from_function(func)</code>	

5.7.2 happily.serialization.deserializer.DeserializerWithSchema

class happily.serialization.deserializer.**DeserializerWithSchema** (*schema*)
Bases: *happily.serialization.deserializer.Deserializer*, *abc.ABC*

schema

5.8 happily.handling.handler

Description

Classes

<i>Handler</i>	A class containing logic to handle a parsed message.
----------------	--

5.8.1 happily.handling.handler.Handler

class happily.handling.handler.**Handler**
Bases: *abc.ABC*

A class containing logic to handle a parsed message.

<i>handle</i> (message)	Applies logic using a provided message, optionally gives back one or more results.
<i>on_handling_failed</i> (message, error)	Applies fallback logic using a provided message when <i>handle ()</i> 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 *Optional[Mapping[str, Any]]*

Returns None if no result is extracted from handling, a dictionary of attributes for single result

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 *Optional[Mapping[str, Any]]*

Returns None if no result is extracted from handling, a dictionary of attributes for single result

5.9 happily.handling.dummy_handler._DummyHandler

class happily.handling.dummy_handler._DummyHandler

Bases: *happily.handling.handler.Handler*

<i>handle</i> (message)	Applies logic using a provided message, optionally gives back one or more results.
<i>on_handling_failed</i> (message, error)	Applies fallback logic using a provided message when <i>handle()</i> 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

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

5.10 happily.exceptions

Description

Exceptions

<i>FetchNoResult</i> ()	Exception thrown by <i>Executor.run_for_result()</i> when it is unable to fetch a result
<i>StopPipeline</i> ([reason])	This exception should be raised to stop a pipeline.

5.10.1 `happyly.exceptions.FetchedNoResult`

exception `happyly.exceptions.FetchedNoResult`

Exception thrown by `Executor.run_for_result()` when it is unable to fetch a result

5.10.2 `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.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

h

- `happyly.caching.cacher`, [17](#)
- `happyly.caching.mixins`, [18](#)
- `happyly.exceptions`, [21](#)
- `happyly.handling.handler`, [20](#)
- `happyly.listening.executor`, [11](#)
- `happyly.listening.listener`, [15](#)
- `happyly.schemas.schema`, [17](#)
- `happyly.serialization.deserializer`, [19](#)
- `happyly.serialization.serializer`, [18](#)

Symbols

`_DummyHandler` (class in `happyly.handling.dummy_handler`), 21

A

`ack()` (`happyly.listening.listener.ListenerWithAck` method), 17

`add()` (`happyly.caching.cacher.Cacher` method), 18

B

`BaseListener` (class in `happyly.listening.listener`), 15

C

`CacheByRequestIdMixin` (class in `happyly.caching.mixins`), 18

`Cacher` (class in `happyly.caching.cacher`), 17

D

`deserialized` (`happyly.listening.executor.ResultAndDeserialized` attribute), 15

`Deserializer` (class in `happyly.serialization.deserializer`), 19

`deserializer` (`happyly.listening.executor.Executor` attribute), 12

`DeserializerWithSchema` (class in `happyly.serialization.deserializer`), 20

E

`EarlyAckListener` (class in `happyly.listening.listener`), 16

`Executor` (class in `happyly.listening.executor`), 11

F

`FetchNoResult`, 22

G

`get()` (`happyly.caching.cacher.Cacher` method), 18

H

`handle()` (`happyly.handling.dummy_handler._DummyHandler` method), 21

`handle()` (`happyly.handling.handler.Handler` method), 20

`Handler` (class in `happyly.handling.handler`), 20

`handler` (`happyly.listening.executor.Executor` attribute), 12

`happyly.caching.cacher` (module), 17

`happyly.caching.mixins` (module), 18

`happyly.exceptions` (module), 21

`happyly.handling.handler` (module), 20

`happyly.listening.executor` (module), 11

`happyly.listening.listener` (module), 15

`happyly.schemas.schema` (module), 17

`happyly.serialization.deserializer` (module), 19

`happyly.serialization.serializer` (module), 18

L

`LateAckListener` (class in `happyly.listening.listener`), 16

`ListenerWithAck` (class in `happyly.listening.listener`), 16

O

`on_acked()` (`happyly.listening.listener.ListenerWithAck` method), 16

`on_deserialization_failed()` (`happyly.listening.executor.Executor` method), 13

`on_deserialized()` (`happyly.listening.executor.Executor` method), 12

`on_finished()` (`happyly.listening.executor.Executor` method), 14

`on_finished()` (`happyly.listening.listener.LateAckListener` method), 16

`on_handled()` (`happyly.listening.executor.Executor` method), 13

`on_handling_failed()` (`happyly.handling.dummy_handler._DummyHandler` method), 21

method), 21
on_handling_failed() (*happyly.handling.handler.Handler method*), 20
on_handling_failed() (*happyly.listening.executor.Executor method*), 13
on_published() (*happyly.listening.executor.Executor method*), 13
on_publishing_failed() (*happyly.listening.executor.Executor method*), 14
on_received() (*happyly.listening.executor.Executor method*), 12
on_stopped() (*happyly.listening.executor.Executor method*), 14

P

publisher (*happyly.listening.executor.Executor attribute*), 12

R

remove() (*happyly.caching.cacher.Cacher method*), 18
result (*happyly.listening.executor.ResultAndDeserialized attribute*), 15
ResultAndDeserialized (*class in happyly.listening.executor*), 14
run() (*happyly.listening.executor.Executor method*), 14

S

Schema (*class in happyly.schemas.schema*), 17
Serializer (*class in happyly.serialization.serializer*), 19
SerializerWithSchema (*class in happyly.serialization.serializer*), 19
StopPipeline, 22
subscriber (*happyly.listening.listener.BaseListener attribute*), 15