event_processor

Release 3.1.4

Nicolas Marier

CONTENTS

This library aims to simplify the common pattern of event processing. It simplifies the process of filtering, dispatching and pre-processing events as well as injecting dependencies in event processors.

The only requirement is that your events are regular python dictionaries.

Take a look at the following examples to get an overview of the features available! Of course, you can mix and combine them in any way you like to create more complex scenarios.

DOCUMENTATION

1.1 Core Concepts

The core idea of this library is really simple. All you have are processors, filters and dependencies. Read on to learn more about each concept.

1.1.1 Processors

Processors are simply functions that you create to process a certain event. Which processor gets called for which event depends on the filters you use for that processor. More info on the *processors* page.

1.1.2 Filters

Filters are how you tell the library which processor to invoke for each event you want to process. There's a few different kinds of filters, the most common being the `Exists` filter and the `Eq` filter. More info on the *filters* page.

Note: It's possible to have ambiguous filters (see *Filters* for details). To resolve them, take a look at *Ranking Processors* and *Invocation Strategies*.

1.1.3 Dependencies

Dependencies are a pretty key concept, because they allow your processors to depend on values obtained dynamically (which is super important if you want to use external APIs in your processors). It's also possible to depend on the event (so you can have it injected into your processor). More info on the *dependencies* page.

1.1.4 Example

This example shows how these three concepts click together to make event processing easy. For the example, we just use a stub for the SSM client and assume that the `admin_email` parameter has a value of `admin@example.com`.

```python
from event_processor import EventProcessor, Event, Depends
from event_processor.filters import Exists


processor = EventProcessor()

```

<div align="right">(continues on next page)</div>

```python
def get_ssm():
    return FakeSSMClient()



@processor(Exists("user.email"))
def user_is_admin(raw_event: Event, ssm_client: FakeSSMClient = Depends(get_ssm)) ->
↪bool:
    ssm_response = ssm_client.get_parameter(Name="admin_email")
    admin_email = ssm_response["Parameter"]["Value"]
    return raw_event["user"]["email"] == admin_email

print("admin@example.com is admin:", processor.invoke({"user": {"email": "admin@example.
↪com"}}).returned_value)
print("user@example.com is admin:", processor.invoke({"user": {"email": "user@example.com
↪"}}).returned_value)
```

```
admin@example.com is admin: True
user@example.com is admin: False
```

You can see that because the event contains a value at `user.email` (i.e. this path `Exists` in the event), the processor was invoked. It also received the event by specifying a parameter with the `Event` type and received an SSM client by depending on the value returned by `get_ssm`.

1.2 Processors

Processors are the least involved part of the library. All you have to do is register your processors into an event processor so that events can be dispatched to it. For a basic example, see the *Core Concepts*.

1.2.1 Parameters

You can't specify just any random parameters for your processors, event-processor needs to know what to do with them when invoking your processor. The parameters that your processor can accept are documented in the *Dependencies* section!

1.2.2 Invocation

When you invoke your event processor, event-processor takes care of running the correct function (or functions, depending on your invocation strategy). After running your function, it packs some information with the returned value into a *Results* object and returns that to the calling code.

With the result, you can get the value returned by your function as well as the name of the processor that was invoked (the name of your function).

The actual return value for the invocation depends on your invocation strategy. Whether you get a single value or a list returned from the invocation should be very obvious from the invocation strategy you're using. Essentially, if the strategy can call multiple processors, you get a list. If not, you get a single value.

1.2.3 Error Handling

Handling errors with a lot of processors can get pretty repetitive, especially if you want to ignore several errors. This might happen when you want to run all matching processors, and you don't want an error in one processor to interrupt the whole processing.

This is why you can use error handling strategies. Here's an example :

```python
from event_processor import EventProcessor, ErrorHandlingStrategies
from event_processor.filters import Accept

processor = EventProcessor(error_handling_strategy=ErrorHandlingStrategies.CAPTURE)


@processor(Accept())
def my_failing_processor():
    raise RuntimeError("Oh no, I failed!")


result = processor.invoke({})

if result.has_exception:
    print(str(result.raised_exception))
```

```
Oh no, I failed!
```

Note: Notice that no exception was raised by `invoke`, and instead a `Result` was returned that contained the raised exception.

Here is a list of error handling strategies and what they do :

Bubble (default)

This strategy will bubble up exceptions to the caller of `invoke`, this is just like if you called the processor yourself without the library, this way you can handle errors however you like.

SpecificBubble

This strategy will only bubble up *some* errors to the caller, so you can capture any exception *except* a few specific ones. This could be used if only critical errors should be bubbled up.

Capture

This strategy will capture errors that occur in processors and include them in the `Result` that is returned. This is especially useful when using the *All Matches* invocation strategy, because it will ensure all processors are run even if some of them raise exceptions.

SpecificCapture

This strategy will only capture *some* errors and let other bubble up to the caller, so it's possible to ignore only a few specific errors instead of all of them.

1.2.4 Multiple Event Processors

Note that when you register a processor, it will be invoked only by the event processor for which it is registered. For example,

```python
from event_processor import EventProcessor, InvocationError
from event_processor.filters import Accept

processor = EventProcessor()
other_processor = EventProcessor()


@processor(Accept())
def my_processor():
    pass


processor.invoke({})  # This is fine, a processor exists for the event

try:
    other_processor.invoke({})  # This will raise
except InvocationError:
    print("Raised!")
```

```
Raised!
```

1.2.5 Sub-Processors

In a big application, you might not want to have all your processors in the same module, so it's possible to setup sub-processors which get merged with a main processor.

Note: You can also add multiple sub-processors in a single function call with the `add_subprocessors()` method. This is really only for convenience and aesthetics, there's no functional difference with calling `add_subprocessor()` multiple times.

`my_module.py`

```python
from event_processor import EventProcessor
from event_processor.filters import Accept

sub_processor = EventProcessor()


@sub_processor(Accept())
def my_processor():
    pass
```

`main.py`

```python
from event_processor import EventProcessor
from event_processor.filters import Accept

# from my_module import sub_processor

main_processor = EventProcessor()
main_processor.add_subprocessor(sub_processor)

# Note that we are invoking on the main processor,
# but the event will be dispatched to the sub-processor.
result = main_processor.invoke({})

print(result.returned_value)
```

```
sub_processing!
```

1.2.6 Package Sub-Processors

When your application grows even further, you might end up with a larger collection of event processors spread across several modules. In this case, it becomes tedious to import each event processor from each of the modules manually. To make it easy to appropriately separate your processors, it's possible to automatically add all the processors found in all the modules contained within a given package.

With the following directory structure, this is how it would work :

```
project-root
└── src
    └── processors
        ├── my_module_1.py
```

```
            ├── my_module_2.py
            └── file4
                └── my_module_3.py
```

```python
from event_processor import EventProcessor

from src import processors

event_processor = EventProcessor()
event_processor.add_subprocessors_in_package(processors)
```

Note: It's important not to just use a package name here, you need to use the actual package that you've imported. Also, this will cause all the modules in the package to be imported, so be mindful of circular imports when using this feature!

1.2.7 Ranking Processors

Note: It's not always necessary to use ranking. Take a look at the warning on the *Filters* page to learn more and see if it's something you need to be concerned about.

Since it's not possible for the library to guess what should happen to a particular event matching multiple filters, figuring that out is left up to the user. In most cases, it's as simple as not worrying about it, but sometimes, dealing with ambiguous filters is just unavoidable.

This is when you should use processor ranking. A processor's rank is basically an indicator of how much priority it has with regards to other processors. It's what helps the library call the right processor for an event that might match multiple processors.

Here's an example of how you can use ranking :

Note: The default rank for processors is `0`. The matching processor with the highest rank will be called. **To learn how to specify what to do when multiple processors match with the same rank, see** *Invocation Strategy*.

Another useful thing to think about is that you can use the `-1` rank to make a processor be called last when there are multiple matches. This is especially useful when coupled with the *Accept* filter.

```python
from event_processor import EventProcessor
from event_processor.filters import Exists, Eq

processor = EventProcessor()


@processor(Exists("a"))
def processor_a():
    print("Processor a!")
```

```python
@processor(Eq("a", "b"), rank=1)
def processor_b():
    print("Processor b!")


processor.invoke({"a": "b"})
processor.invoke({"a": "not b"})
```

```
Processor b!
Processor a!
```

1.2.8 Invocation Strategy

To choose how to invoke your processor(s) in the case that multiple processors with the same rank all match a given event, you have to choose an invocation strategy.

Note: The default invocation strategy is the *First Match* strategy.

First Match

This strategy calls the first matching processor (among those with the highest rank). It returns a simple *Result*.

All Matches

This strategy calls all the matching processors (that have the highest rank). It returns a list of *Result*, one for each matching processor (even if only a single match occurred).

No Matches

This strategy calls none of the matching processors if there are more than one (and returns a *Result* with a None value). Otherwise, it calls the single matching processor and returns a *Result* with that result.

No Matches Strict

This strategy calls none of the matching processors if there are more than one, and it raises an exception. Otherwise, it calls the single matching processors and returns a *Result* with the returned value.

Example

To use a non-default invocation strategy, use the provided `InvocationStrategies` enum like so :

```python
from event_processor import EventProcessor, InvocationStrategies
from event_processor.filters import Exists, Eq

processor = EventProcessor(invocation_strategy=InvocationStrategies.ALL_MATCHES)


@processor(Exists("a"))
def processor_a():
    print("Processor a!")


@processor(Eq("a", "b"))
def processor_b():
    print("Processor b!")


processor.invoke({"a": "b"})
```

```
Processor a!
Processor b!
```

1.3 Filters

There are a few available filters to help you make sure the correct processor is invoked for the correct event. To see how to use filters in practice, see the *core concepts*.

> **Warning:** It's possible to create different filters that will match the same event. For example, when using the *Exists* and *Eq* filters on the same key, if the *Eq* filter matches, then the *Exists* filter is guaranteed to match.
>
> Have a look at *Ranking Processors* to learn how to resolve these ambiguities. Also, note that these issues may not apply to your context. You only have to worry about this if you have ambiguous filters.
>
> Also, you may actually want to run multiple processors for the same event. If this is the case, then you should look at *Invocation Strategies* instead of ranking.

1.3.1 Accept

This filter will always match any event it is presented with. It will even match things that are not dictionaries. Use this if you need to take a default action whenever no processor exists for an event, or if an unexpected event was sent to your system.

```python
from event_processor.filters import Accept

accept = Accept()

print(accept.matches({}))
```

(continues on next page)

```
print(accept.matches(None))
print(accept.matches({"Hello", "World"}))
```

```
True
True
True
```

1.3.2 Exists

This filter matches events that contain a certain key (which can be nested), but the value can be anything.

```
from event_processor.filters import Exists

a_exists = Exists("a")
nested = Exists("a.b.c")

print(a_exists.matches({"a": None}))
print(a_exists.matches({"a": 2}))
print(a_exists.matches({}))
print(nested.matches({"a": {"b": {"c": None}}}))
print(nested.matches({"a": {"b": {"c": 0}}}))
```

```
True
True
False
True
True
```

1.3.3 Eq

This filter matches a subset of the events matched by *Exists*. It only matches the events where a specific value is found at the specified key (as opposed to just existing).

```
from event_processor.filters import Eq

a_is_b = Eq("a", "b")
a_b_c_is_none = Eq("a.b.c", None)

print(a_is_b.matches({"a": "b"}))
print(a_is_b.matches({"a": 2}))
print(a_b_c_is_none.matches({"a": {"b": {"c": None}}}))
print(a_b_c_is_none.matches({"a": {"b": {"c": 0}}}))
```

```
True
False
True
False
```

1.3.4 NumCmp

This filter matches numbers that satisfy a comparison function with a given target.

Note: You should try to avoid using this filter directly and instead use *Lt, Leq, Gt, Geq* when possible.

The reason for this advisory is that in python, callables with the same code will compare as not being equal, which means that if you start using lambdas as the comparator (and more critically, if you use different lambdas that have the same behavior as comparators), then the equality checks for this filter will be inaccurate. This leads to duplicate processors not raising exceptions at import time.

The tl;dr: if you use this filter, don't use lambdas as comparators and don't use different functions that do the same thing either.

```python
from event_processor.filters import NumCmp

def y_greater_than_twice_x(x, y):
    return (2 * x) < y

# Note that the comparator is the same here, this is important.
# You can use different comparators, but only if they do different things.
twice_a_less_than_four = NumCmp("a", y_greater_than_twice_x, 4)
twice_a_less_than_eight = NumCmp("a", y_greater_than_twice_x, 8)

print(twice_a_less_than_four.matches({"a": 1}))
print(twice_a_less_than_four.matches({"a": 2}))
print(twice_a_less_than_eight.matches({"a": 3}))
print(twice_a_less_than_eight.matches({"a": 4}))
print(twice_a_less_than_eight.matches({"not-a": 2}))
```

```
True
False
True
False
False
```

1.3.5 Lt, Leq, Gt, Geq

These filters all work in the same way in that they match when a value is present at the given path and it satisfies a comparison operation.

- Lt means <
- Leq means <=
- Gt means >
- Geq means >=

```python
from event_processor.filters import Lt, Leq, Gt, Geq

a_lt_0 = Lt("a", 0)
a_leq_0 = Leq("a", 0)
```

```python
a_gt_0 = Gt("a", 0)
a_geq_0 = Geq("a", 0)

print(a_lt_0.matches({"a": 0}))
print(a_leq_0.matches({"a": 0}))
print(a_gt_0.matches({"a": 0}))
print(a_geq_0.matches({"a": 0}))
```

```
False
True
False
True
```

1.3.6 Dyn

This filter accepts a resolver parameter, which is any callable. Whether or not it matches a given event depends on the return value of the resolver. If the resolver returns a truthy value, then the filter matches. Otherwise, it doesn't. This is useful when your events have a more complex structure that can't really be handled by other existing filters.

> **Warning:** When using a dynamic filter, it's your job to make sure the functions you supply won't match the same events (and if they do, to specify a *rank* or an *invocation strategy*).

With the Dyn filter, it's useful to use lambda functions because they fit nicely in one line and won't clutter your code. If you use lambda functions, the functions you create **must** accept a single argument (which will be the event).

```python
from event_processor.filters import Dyn

a_len_is_0 = Dyn(lambda e: len(e.get("a", [])) == 0)
a_len_is_bigger = Dyn(lambda e: len(e.get("a", [])) >= 1)

print(a_len_is_0.matches({"a": []}))
print(a_len_is_0.matches({"a": [0]}))
print(a_len_is_bigger.matches({"a": []}))
print(a_len_is_bigger.matches({"a": [0, 1]}))
```

```
True
False
False
True
```

It's also possible to use standard functions with the Dyn filter, in which case you can specify any argument that would be valid for a dependency (see *Dependencies* for details). For example :

```python
from event_processor import Depends, Event
from event_processor.filters import Dyn

def my_dependency():
    return 0
```

```python
def my_filter_resolver(event: Event, dep_value: int = Depends(my_dependency)):
    return event["key"] == dep_value

a_filter = Dyn(my_filter_resolver)

print(a_filter.matches({"key": 0}))
print(a_filter.matches({"key": 1}))
```

```
True
False
```

1.3.7 And

This filter does exactly what you would expect, and matches when all the events supplied to it as arguments match. It acts as a logical AND between all its sub-filters.

```python
from event_processor.filters import And, Exists

a_exists = Exists("a")
b_exists = Exists("b")
c_exists = Exists("c")

a_and_b_exist = And(a_exists, b_exists)
a_b_and_c_exist = And(a_exists, b_exists, c_exists)

print(a_and_b_exist.matches({"a": 0, "b": 0}))
print(a_and_b_exist.matches({"a": 0, "b": 0, "c": 0}))
print(a_b_and_c_exist.matches({"a": 0, "b": 0}))
print(a_b_and_c_exist.matches({"a": 0, "b": 0, "c": 0}))
```

```
True
True
False
True
```

You can also use & between processors instead of And explicitly to make your filters prettier.

```python
from event_processor.filters import And, Exists

a_exists = Exists("a")
b_exists = Exists("b")
c_exists = Exists("c")

a_and_b_exist = a_exists & b_exists
a_b_and_c_exist = a_exists & b_exists & c_exists

print(a_and_b_exist.matches({"a": 0, "b": 0}))
print(a_and_b_exist.matches({"a": 0, "b": 0, "c": 0}))
print(a_b_and_c_exist.matches({"a": 0, "b": 0}))
print(a_b_and_c_exist.matches({"a": 0, "b": 0, "c": 0}))
```

```
True
True
False
True
```

1.3.8 Or

This filter is similar to the *And* filter, except that it will match if any of its sub-filters match.

```python
from event_processor.filters import Or, Exists

a_exists = Exists("a")
b_exists = Exists("b")
c_exists = Exists("c")

a_b_or_c_exist = Or(a_exists, b_exists, c_exists)

print(a_b_or_c_exist.matches({"a": 0}))
print(a_b_or_c_exist.matches({"b": 0}))
print(a_b_or_c_exist.matches({"c": 0}))
print(a_b_or_c_exist.matches({"d": 0}))
```

```
True
True
True
False
```

Again, to make things more ergonomic, you can use | instead of Or.

```python
from event_processor.filters import Or, Exists

a_exists = Exists("a")
b_exists = Exists("b")
c_exists = Exists("c")

a_b_or_c_exist = a_exists | b_exists | c_exists

print(a_b_or_c_exist.matches({"a": 0}))
print(a_b_or_c_exist.matches({"b": 0}))
print(a_b_or_c_exist.matches({"c": 0}))
print(a_b_or_c_exist.matches({"d": 0}))
```

```
True
True
True
False
```

1.4 Dependencies

Dependency injection is a useful tool that you can use to keep your code clean and testable, which is why this library offers simple dependency injection out of the box. The current offering was heavily inspired by the excellent FastAPI framework.

1.4.1 Functional Dependencies

This type of dependency is the most flexible and powerful. It essentially allows you to inject a value into your processor which will be computed from the result of another function of your choice.

Note: These dependencies are cached by default, so if that's something you don't want, be sure to specify `cache=False` in your dependency.

> **Warning:** Caching happens per-invocation, so if a sub-dependency is used by many dependencies, it will only be resolved once per invocation. If you invoke the same processor again, the dependency will be invoked again.
>
> If you want something to be cached permanently, you should look at python's cache decorator.

Simple Example

```python
from event_processor import EventProcessor, Depends
from event_processor.filters import Accept

processor = EventProcessor()


def get_my_value():
    return 42


@processor(Accept())
def my_processor(my_value : int = Depends(get_my_value)):
    print(my_value)


processor.invoke({})
```

```
42
```

Nesting Example

You can also nest dependencies as deep as you want to go, so you can easily re-use them.

```python
from event_processor import EventProcessor, Depends
from event_processor.filters import Accept

processor = EventProcessor()


def get_zero():
    return 0


# This dependency can itself depend on another value
def get_my_value(zero: int = Depends(get_zero)):
    return zero + 1


@processor(Accept())
def my_processor_with_caching(my_value : int = Depends(get_my_value)):
    print(my_value)


processor.invoke({})
```

```
1
```

Class Dependencies

Classes themselves are also callables. By default, their init method will be called when you call them, so you can use classes as dependencies as well.

```python
from event_processor import EventProcessor, Depends, Event
from event_processor.filters import Exists

processor = EventProcessor()


class MyThing:
    def __init__(self, event: Event):
        self.username = event["username"]

    def get_username(self):
        return self.username


@processor(Exists("username"))
```

```python
def my_processor_with_caching(my_thing : MyThing = Depends(MyThing)):
    print(my_thing.get_username())


processor.invoke({"username": "someone"})
```

```
someone
```

1.4.2 Event Dependencies

Sometimes it's useful for processors to receive a copy of the event that triggered their invocation, so you can easily signal that it is required by your processor by having a parameter annotated with the Event type.

Note: Event dependencies follow the same rules as other dependencies in that other dependencies can depend on the event, allowing dynamic fetching of data or just creation of a convenient type for the event.

Here's an example of a simple event dependency :

```python
from event_processor import EventProcessor, Event
from event_processor.filters import Accept

processor = EventProcessor()


@processor(Accept())
def my_processor_with_caching(event: Event):
    print(event)


processor.invoke({"hello": "world"})
```

```
{'hello': 'world'}
```

And here's an example where a dependency depends on the event :

```python
from event_processor import EventProcessor, Event
from event_processor.filters import Exists

processor = EventProcessor()


# This function could also query a database (in which case it might depend
# on another function that will return a connection from a connection pool).
def extract_email(event: Event):
    return event["email"]


@processor(Exists("email"))
def my_processor_with_caching(email: str = Depends(extract_email)):
```

```
    print(email)


processor.invoke({"email": "someone@example.com"})
```

```
someone@example.com
```

1.4.3 Pydantic Dependencies

Pydantic is a library which helps with data validation and settings management using python type annotations. You can leverage it in event processors to benefit from both the convenience of automatically parsing an event into a given type and having it fully validated. Pydantic can also provide detailed and friendly error messages to users for validation errors.

Here's a simple example to illustrate how the event might be parsed for use in a processor :

```python
from event_processor import EventProcessor
from event_processor.filters import Eq
from pydantic import BaseModel

processor = EventProcessor()


class CreateUserQuery(BaseModel):
    email: str
    password: str


@processor(Eq("query", "create_user"))
def handle_user_creation(query: CreateUserQuery):
    print(query.email)
    print(query.password)


processor.invoke(
    {"query": "create_user", "email": "someone@example.com", "password": "hunter2"}
)
```

```
someone@example.com
hunter2
```

You can also add custom validations for fields using validators as well as many other things. Take a look at the pydantic docs to learn more!

1.4.4 Scalar Dependencies

Sometimes, you don't need many parts of an input event, just one or two fields, so depending on the whole event or having to make a pydantic model just for a few fields might feel excessive. This is what scalar dependencies are good for.

> **Warning:** If you want to benefit from type validation for your scalar dependencies, you need to have pydantic installed. If you don't have pydantic, no types will be validated for scalar dependencies (really, not even basic ones).
>
> Also, if you *do* use pydantic, but don't specify a type annotation for a parameter, then `typing.Any` is assumed.

Here's a very basic example :

```python
from event_processor import EventProcessor
from event_processor.filters import Exists

processor = EventProcessor()


@processor(Exists("email"))
def handle_user(email: str):
    print(email)


processor.invoke({"email": "someone@example.com"})
```

```
someone@example.com
```

Here's an example with a pydantic field type :

```python
from event_processor import EventProcessor
from event_processor.filters import Exists
from pydantic import ValidationError
from pydantic.color import Color

processor = EventProcessor()


@processor(Exists("my_color"))
def handle_user(my_color: Color):
    print(my_color.as_hex())


processor.invoke({"my_color": "white"})

try:
    processor.invoke({"my_color": "not-a-color"})
except ValidationError as e:
    print(e.errors()[0]["msg"])
```

```
#fff
value is not a valid color: string not recognised as a valid color
```

1.5 Testing Processors

Thanks to the separation between the definition and invocation of processors, it's really easy to test processors. Since the processor decorator returns the function as-is (and does not modify it), it's possible to test your processor the same way you would test any other function. Dependencies also make it easy to use mocks for your external service dependencies.

Here's an example of how you might test a processor :

```python
from event_processor import EventProcessor, Event, Depends
from event_processor.filters import Exists

processor = EventProcessor()


class FakeDatabase:
    values = {
        "users": [
            {"email": "admin@example.com", "role": "admin"},
            {"email": "user@example.com", "role": "user"},
        ]
    }

    def get_role_by_email(self, email: str) -> str:
        user = [user for user in self.values["users"] if user["email"] == email][0]
        return user["role"]


database_instance = None
def get_database() -> FakeDatabase:
    global database_instance
    if database_instance is None:
        database_instance = FakeDatabase()
    return database_instance


def extract_email(event: Event):
    return event["email"]


@processor(Exists("email"))
def user_is_admin(
    email: str = Depends(extract_email, cache=False),
    db_client: FakeDatabase = Depends(get_database),
):
    user_role = db_client.get_role_by_email(email)
    return user_role == "admin"
```

```python
print(processor.invoke({"email": "user@example.com"}).returned_value)
print(processor.invoke({"email": "admin@example.com"}).returned_value)


#################### Tests ####################
from unittest.mock import Mock


def test_user_is_admin_returns_true_for_admin_user():
    mock_db = Mock()
    mock_db.get_role_by_email.return_value = "admin"

    result = user_is_admin("someone@example.com", mock_db)

    assert result is True


def test_user_is_admin_returns_false_for_non_admin_user():
    mock_db = Mock()
    mock_db.get_role_by_email.return_value = "user"

    result = user_is_admin("someone@example.com", mock_db)

    assert result is False


test_user_is_admin_returns_true_for_admin_user()
test_user_is_admin_returns_false_for_non_admin_user()
```

```
False
True
```

1.6 API Documentation

1.6.1 Event Processor

class `src.event_processor.event_processor.`**EventProcessor**(*invocation_strategy:* InvocationStrategies *= InvocationStrategies.FIRST_MATCH, error_handling_strategy: ErrorHandlingStrategies = ErrorHandlingStrategies.BUBBLE*)

A self-contained event processor.

add_subprocessor(*subprocessor:* EventProcessor)

Add a subprocessor to this event processor

Parameters

subprocessor – The other event processor to add

add_subprocessors(**subprocessors:* EventProcessor)

>   Add multiple subprocessors at once.

>   > **Parameters**
>   >> **subprocessors** – The tuple of subprocessors

add_subprocessors_in_package(*package: module*)

>   Add all the processors found in all modules of a package as subprocessors.

>   Note that you should specify an actual package, and not just the name of a package.

>   > **Parameters**
>   >> **package** – The package that should be searched for event processors

invoke(*event: Dict*) → Union[*Result*, List[*Result*]]

>   Invoke the correct processor for an event.

>   There may be multiple processors invoked, depending on the invocation strategy.

>   > **Parameters**
>   >> **event** – The event to find a processor for

>   > **Returns**
>   >> The return value of the processor

processor(*event_filter:* Filter, *rank: int = 0*)

>   Register a new processor with the given filter and rank.

>   > **Parameters**
>   >> - **event_filter** – The filter for which to match events
>   >> - **rank** – This processor's rank (when there are multiple matches for a single event)

1.6.2 Invocation Strategies

Contains the different invocation strategies for calling processors.

class src.event_processor.invocation_strategies.**AllMatches**(*error_handling_strategy:*
~src.event_processor.error_handling_strategies.ErrorHan
=
<src.event_processor.error_handling_strategies.Bubble
object>)

>   Strategy calling all matching processors.

>   **invoke**(*matching: List[Callable]*, *event: Optional[*Event*] = None*, *cache: Optional[Dict] = None*) →
>   List[*Result*]

>   > Invoke one or multiple matching processors.

class src.event_processor.invocation_strategies.**FirstMatch**(*error_handling_strategy:*
~src.event_processor.error_handling_strategies.ErrorHan
=
<src.event_processor.error_handling_strategies.Bubble
object>)

>   Strategy calling the first matching processor.

>   **invoke**(*matching: List[Callable]*, *event: Optional[*Event*] = None*, *cache: Optional[Dict] = None*) → *Result*

>   > Invoke one or multiple matching processors.

class src.event_processor.invocation_strategies.**InvocationStrategies**(*value*)

    Enumeration of available invocation strategies.

class src.event_processor.invocation_strategies.**InvocationStrategy**(*error_handling_strategy:*
~src.event_processor.error_handling_strategies
=
<src.event_processor.error_handling_strategies
object>)

    Class defining an abstract invocation strategy.

    abstract invoke(*matching: List[Callable]*, *event: Optional[*Event*] = None*, *cache: Optional[Dict] =*
        None) → Union[*Result*, List[*Result*]]

        Invoke one or multiple matching processors.

class src.event_processor.invocation_strategies.**NoMatches**(*error_handling_strategy:*
~src.event_processor.error_handling_strategies.ErrorHand
=
<src.event_processor.error_handling_strategies.Bubble
object>)

    Strategy not calling any matching processors.

    invoke(*matching: List[Callable]*, *event: Optional[*Event*] = None*, *cache: Optional[Dict] = None*) → *Result*

        Invoke one or multiple matching processors.

class src.event_processor.invocation_strategies.**NoMatchesStrict**(*error_handling_strategy:*
~src.event_processor.error_handling_strategies.Err
=
<src.event_processor.error_handling_strategies.Bu
object>)

    Strategy failing when there are multiple matching.

    invoke(*matching: List[Callable]*, *event: Optional[*Event*] = None*, *cache: Optional[Dict] = None*) → *Result*

        Invoke one or multiple matching processors.

1.6.3 Results

class src.event_processor.result.**Result**(*processor_name: str*, *returned_value: Optional[Any] = None*,
    raised_exception: Optional[Exception] = None)

    A result is what gets returned after an invocation.

    It contains information about the processor as well as its return value.

1.6.4 Exceptions

Exceptions for event processor.

exception src.event_processor.exceptions.**DependencyError**

    Exceptions for failures while resolving dependencies.

exception src.event_processor.exceptions.**EventProcessorError**

    General exception for the event-processor library.

exception src.event_processor.exceptions.**FilterError**

    Exception for failures related to filters.

exception src.event_processor.exceptions.**InvocationError**

> Exception for failures in invocation.

exception src.event_processor.exceptions.**NoValueError**

> Exception for when a value is not present in a given context.

1.6.5 Filtering

Contains many different filters to conveniently filter through events.

class src.event_processor.filters.**Accept**

> Accept any event (good for default processors).
>
> **matches**(*_event: dict*) → bool
>
> > Test whether a given event matches an input event.
> >
> > > **Parameters**
> > > > **event** – The event to test
> > >
> > > **Returns**
> > > > True if the event matches, False otherwise

class src.event_processor.filters.**And**(**args:* Filter)

> Accept events that get accepted by all specified filters.
>
> **matches**(*event: dict*) → bool
>
> > Test whether a given event matches an input event.
> >
> > > **Parameters**
> > > > **event** – The event to test
> > >
> > > **Returns**
> > > > True if the event matches, False otherwise

class src.event_processor.filters.**Dyn**(*resolver: Callable*)

> Accept events based on a dynamic condition which is resolved by a callable.
>
> Note that the equality check with this filter is a bit special. The filter will only be equal if the resolver is the same (two functions with the same code are not equal, they need to be the same object in memory).
>
> This means that when using this filter, you should be careful to either re-use the same function as a resolver or make sure the functions have different behaviors. Otherwise, be sure to use processor ranking and invocation strategies.
>
> **matches**(*event: dict*) → bool
>
> > Test whether a given event matches an input event.
> >
> > > **Parameters**
> > > > **event** – The event to test
> > >
> > > **Returns**
> > > > True if the event matches, False otherwise

class src.event_processor.filters.**Eq**(*path: Any*, *value: Any*)

> Accept events where a given value is present at the given key.
>
> **matches**(*event: dict*) → bool
>
> > Test whether a given event matches an input event.

> > > **Parameters**
> > > > **event** – The event to test
> > >
> > > **Returns**
> > > > True if the event matches, False otherwise

class `src.event_processor.filters.`**`Exists`**(*path: Any*)

> Accept event where a given key exists.
>
> > **matches**(*event: dict*) → bool
> >
> > > Test whether a given event matches an input event.
> > >
> > > > **Parameters**
> > > > > **event** – The event to test
> > > >
> > > > **Returns**
> > > > > True if the event matches, False otherwise

class `src.event_processor.filters.`**`Filter`**

> Abstract filter to define the filter interface.
>
> > **abstract matches**(*event: dict*) → bool
> >
> > > Test whether a given event matches an input event.
> > >
> > > > **Parameters**
> > > > > **event** – The event to test
> > > >
> > > > **Returns**
> > > > > True if the event matches, False otherwise

class `src.event_processor.filters.`**`Geq`**(*path: Any*, *value: Union[int, float]*)

> Accept events where the value exists and is greater than or equal to the specified value.

class `src.event_processor.filters.`**`Gt`**(*path: Any*, *value: Union[int, float]*)

> Accept events where the value exists and is greater than the specified value.

class `src.event_processor.filters.`**`Leq`**(*path: Any*, *value: Union[int, float]*)

> Accept events where the value at the given path exists is less than or equal to the specified value.

class `src.event_processor.filters.`**`Lt`**(*path: Any*, *value: Union[int, float]*)

> Accept events where the value at the given path exists and is less than the specified value.

class `src.event_processor.filters.`**`NumCmp`**(*path: Any*, *comparator: Callable[[float, float], bool]*, *target: float*)

> Accept events when the comparator returns True.
>
> If you use this processor, make sure that you don't use equal (and not identical) comparators for the same path. For example, don't use the same lambda in two different places. Instead, use a function, and pass a reference to that function. If you don't do that, the filters will effectively be different (even if they match the same thing), leading to perhaps unexpected results.
>
> > **matches**(*event: dict*) → bool
> >
> > > Test whether a given event matches an input event.
> > >
> > > > **Parameters**
> > > > > **event** – The event to test
> > > >
> > > > **Returns**
> > > > > True if the event matches, False otherwise

class src.event_processor.filters.**Or**(*args:* Filter)

> Accept events that get accepted by at least one specified filter.

> **matches**(*event: dict*) → bool

>> Test whether a given event matches an input event.

>>> **Parameters**
>>> **event** – The event to test

>>> **Returns**
>>> True if the event matches, False otherwise

1.6.6 Dependency Injection

Dependency injection and management facilities.

class src.event_processor.dependencies.**Depends**(*callable_: Callable, cache: bool = True*)

> Class to designate a dependency

class src.event_processor.dependencies.**Event**(*dict_event: dict*)

> Type to wrap a dict to be used as a dependency.

src.event_processor.dependencies.**call_with_injection**(*callable_: Callable, event: Optional[*Event*] = None, cache: Optional[dict] = None*) → Optional[Any]

> Call a callable and inject required dependencies.

> Note that keyword args that have the same name as the parameter used for a dependency will be overwritten with the dependency's injected value.

>> **Parameters**

>>> • **callable** – The callable to call

>>> • **event** – The event for the current invocation

>>> • **cache** – The dependency cache to use

>> **Returns**
>> The return value of the callable

src.event_processor.dependencies.**get_event_dependencies**(*callable_: Callable*) → List[str]

> Get the parameter names for event dependencies.

>> **Parameters**
>> **callable** – The callable for which to get dependencies

>> **Returns**
>> A list of the parameters requiring the event

src.event_processor.dependencies.**get_pydantic_dependencies**(*callable_: Callable*) → Dict[str, Type[BaseModel]]

> Get the required models and their parameter names for a callable.

>> **Parameters**
>> **callable** – The callable for which to get dependencies

>> **Returns**
>> A mapping of argument names to pydantic model types

src.event_processor.dependencies.**get_required_dependencies**(*callable_: Callable*) → Dict[str, *[Depends](#)*]

> Get the required dependencies for a callable.
>
> > **Parameters**
> > > **callable** – The callable for which to get dependencies
> >
> > **Returns**
> > > A mapping of callable argument names to dependencies

src.event_processor.dependencies.**get_scalar_value_dependencies**(*callable_: Callable*) → List[Parameter]

> Get the scalar value dependencies for a callable.
>
> > **Parameters**
> > > **callable** – The callable for which to get dependencies
> >
> > **Returns**
> > > A view of the parameters that represent dependencies

src.event_processor.dependencies.**resolve**(*dependency: [Depends](#)*, *event: Optional[[Event](#)] = None*, *cache: Optional[dict] = None*) → Tuple[Optional[Any], bool]

> Resolve a dependency into a value.
>
> The resulting values from dependencies are cached and re-used if a cache is supplied and the dependency itself does not explicitly state that it does not want to be cached. Also, any dependency that depends on another dependency where caching has been disabled will also not be cached (because the sub-value may change, which may in turn change the value of the current dependency).
>
> > **Parameters**
> > > - **dependency** – The dependency to resolve
> > > - **event** – The event for the current invocation
> > > - **cache** – The cache for previously resolved dependencies
> >
> > **Returns**
> > > The tuple (resolved_value, cacheable)
> >
> > **Raises**
> > > pydantic.error_wrappers.ValidationError if the event cannot be parsed into a pydantic model

src.event_processor.dependencies.**resolve_scalar_value_dependencies**(*scalar_dependencies: List[Parameter]*, *event: Optional[[Event](#)]*) → Dict[str, Any]

> Resolve the scalar dependencies to values contained in the event.
>
> Values will be resolved differently depending on whether or not pydantic is installed.
>
> > **Parameters**
> > > - **scalar_dependencies** – The dependencies to resolve
> > > - **event** – The event from which to get values
> >
> > **Returns**
> > > A new dict with resolved dependency values

src.event_processor.dependencies.**resolve_scalar_value_dependencies_with_pydantic**(*scalar_dependencies:*
List[Parameter],
event:
Event) →
Dict[str,
Any]

Resolve the scalar dependencies to values contained in the event with pydantic.

This function does validation for the types of values passed into the event. Since this uses pydantic, it's possible to use any pydantic types such as PaymentCardNumber, for example.

> **Parameters**
>
> - **scalar_dependencies** – The dependencies to resolve
>
> - **event** – The event from which to get values
>
> **Returns**
> A new dict with resolved and validated dependency values

src.event_processor.dependencies.**resolve_scalar_value_dependencies_without_pydantic**(*scalar_dependencies:*
List[Parameter],
event:
Event)
→
Dict[str,
Any]

Resolve the scalar dependencies to values contained in the event without using pydantic.

This function does not validate the types of values passed into the event to ensure they match the type annotations of the dependencies. To get validation for those types, make sure pydantic is installed.

> **Parameters**
>
> - **scalar_dependencies** – The dependencies to resolve
>
> - **event** – The event from which to get values
>
> **Returns**
> A new dict with resolved dependency values

CHANGELOG

- v3.1.4: Simplify calling processors by making `EventProcessor` callable.

- v3.1.3: Fix a bug with invocation when an event matches several filters.

- v3.1.2: Fix an incorrect import.

- v3.1.1: Relax pydantic requirements to help dependency management for downstream projects.

- v3.1.0: Add error handling strategies for processors.

- v3.0.1: Improve the import path for results.

- **v3.0.0:**

    - BREAKING CHANGE: The value returned by invoking a processor is now a Result.

    - BREAKING CHANGE: Dependencies are now only cached per-invocation.

    - It's not possible to use identical or equivalent filters for different processors.

- v2.6.0: Support getting a list of invoked processors after an invocation

- v2.5.0: Support adding multiple subprocessors at once and also adding all processors from all modules in a package

- v2.4.1: Fix scalar dependency resolution without pydantic (only raise on actual missing values and not none values)

- v2.4.0: Support scalar value dependencies in processor parameters

- v2.3.1: Raise the correct exception when processor parameters are invalid due to optional args

- v2.3.0: Support dynamic filters

- v2.2.0: Support pydantic models as processor dependencies

- v2.1.1: Fix negative ranks and document the -1 rank usage

- v2.1.0: Add number comparison filters

- v2.0.0: Refactor a lot of the internals, make filters more user friendly and dependency injection more intuitive

- v1.1.0: Add support for subprocessors

- v1.0.0: Move the state and decorators inside a class

- v0.0.1: Initial release

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX