event_processor

Release 1.0.0

Nicolas Marier

May 06, 2021

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.

CHAPTER

ONE

USE-CASES

This library is very generic and it applies to several different problems in several different domains, so here are some use-cases for it. Hopefully this might give you an idea for how the library is applicable to your own use-cases.

1.1 FaaS (AWS, GCP, Azure)

This library is very useful in cloud computing environments where functions as a service are used, such with AWS Lambda, Google Cloud Functions or Azure Functions. These platforms are frequently used to manage the cloud account. For example, by running functions when a resource is launched, or simply on a schedule. They're also used for monitoring. In most cases, functions will take an event as input and should take different actions based on the value of that event.

event-processor is helpful with the architecture of such functions, because it allows easily forwarding events to the right function for processing.

1.1.1 AWS - CloudWatch Alarm -> SNS -> Lambda

```python
import json

from event_processor import EventProcessor


event_processor = EventProcessor()


# Truncated for readability
cloudwatch_event = {
    "id": "c4c1c1c9-6542-e61b-6ef0-8c4d36933a92",
    "detail-type": "CloudWatch Alarm State Change",
    "detail": {
        "alarmName": "ServerCpuTooHigh",
        "previousState": {
            "value": "OK"
        },
        "state": {
            "value": "ALARM"
        }
    }
}

lambda_event = {
```

(continues on next page)

(continued from previous page)

```python
  "Records": [
    {
      "Sns": {
        "Subject": "TestInvoke",
        "Message": json.dumps(cloudwatch_event)
      }
    }
  ]
}


class DummySlackClient:
    def send_message(self, message: str):
        print(f"Send in slack: {message}")


@event_processor.dependency_factory
def messaging_service(service: str):
    if service == "slack":
        return DummySlackClient()
    else:
        raise NotImplementedError()


@event_processor.processor(
    {"detail.previousState.value": "OK", "detail.state.value": "ALARM"},
    messaging_service=("slack",)
)
def process_started_alarming(event, slack_client: DummySlackClient):
    slack_client.send_message(f"Alarm {event['detail']['alarmName']} went from OK to␣
↪ALARM")


@event_processor.processor({}, messaging_service=("slack",))  # Default processor
def default_processor(event, slack_client: DummySlackClient):
    slack_client.send_message(f"Unexpected event: {event}")


def demo_lambda_main(event, _context):
    # You can pre-process events as much as you like before calling invoke.
    # You could also just process the whole raw event, but then filters would
    # be less useful since they couldn't filter against the cloudwatch event.
    # It would also be possible to just update the raw event by parsing the
    # json contained within, so it would be possible to filter on anything.

    cloudwatch_event = json.loads(event["Records"][0]["Sns"]["Message"])
    event_processor.invoke(cloudwatch_event)


demo_lambda_main(lambda_event, {})
lambda_event["Records"][0]["Sns"]["Message"] = json.dumps({"Unexpected": "Oops!"})
demo_lambda_main(lambda_event, {})
```

```
Send in slack: Alarm ServerCpuTooHigh went from OK to ALARM
Send in slack: Unexpected event: {'Unexpected': 'Oops!'}
```

CHAPTER TWO

DOCUMENTATION

2.1 Quick Start

Here are some examples to illustrate common features of the library. These examples are not exhaustive, so you are encouraged to still look through the rest of the docs to discover more powerful or less common use-cases.

2.1.1 Simple Filtering

This is as simple as it gets, just calling the right processor depending on the event.

```python
from typing import Dict

from event_processor import EventProcessor


event_processor = EventProcessor()


@event_processor.processor({"service.type": "service_a"})
def process_service_a(event: Dict):
    return event["service"]["status"] == "up"

@event_processor.processor({"service.type": "service_b"})
def process_service_b(event: Dict):
    return event["service"]["authorized"]

service_a_event = {
    "service": {
        "type": "service_a",
        "status": "down"
    }
}
service_b_event = {
    "service": {
        "type": "service_b",
        "authorized": False
    }
}

print(event_processor.invoke(service_a_event), event_processor.invoke(service_b_
↪event))
```

```
False False
```

2.1.2 Any Filter

Sometimes you want to make sure there's a value at a given path in the event, but you don't care what it is, or you may want to dynamically do things with it in the processor.

```
from typing import Any, Dict

from event_processor import EventProcessor


event_processor = EventProcessor()


@event_processor.processor({"user.email": Any})
def process_user(event: Dict):
    return event["user"]["email"] == "admin@example.com"

print(
    event_processor.invoke({"user": {"email": "admin@example.com"}}),
    event_processor.invoke({"user": {"email": "not-admin@example.com"}})
)
```

```
True False
```

2.1.3 Pre-Processing

It can be convenient to to work with actual python objects rather than raw dictionaries, so you can use pre-processors for processors.

```
from dataclasses import dataclass
from typing import Any, Dict

from event_processor import EventProcessor


event_processor = EventProcessor()


database = {
    "user@example.com": {"role": "user"},
    "admin@example.com": {"role": "admin"}
}

@dataclass
class User:
    email: str
    role: str

def event_to_user(event: Dict):
    email = event["user"]["email"]
    role = database.get(email, {}).get("role")
```

(continues on next page)

(continued from previous page)

```
    return User(email=email, role=role)

@event_processor.processor({"user.email": Any}, pre_processor=event_to_user)
def process_user(user: User):
    return user.role == "admin"

print(
    event_processor.invoke({"user": {"email": "user@example.com"}}),
    event_processor.invoke({"user": {"email": "admin@example.com"}})
)
```

```
False True
```

2.1.4 Dependency Injection

Sometimes, you might want to call external services from a processor, so you can have your dependencies automatically injected.

```
from typing import Any

from event_processor import EventProcessor


event_processor = EventProcessor()


class FakeBotoClient:
    parameters = {"admin_email": "admin@example.com"}

    def get_parameter(self, Name=""):
        return {"Parameter": {"Value": self.parameters.get(Name)}}

@event_processor.dependency_factory
def boto3_clients(client_name: str):
    if client_name == "ssm":
        return FakeBotoClient()
    else:
        raise NotImplementedError()

@event_processor.processor({"user.email": Any}, boto3_clients=("ssm",))
def process_user(event: Dict, ssm_client: FakeBotoClient):
    ssm_response = ssm_client.get_parameter(Name="admin_email")
    admin_email = ssm_response["Parameter"]["Value"]
    return event["user"]["email"] == admin_email

print(
    event_processor.invoke({"user": {"email": "admin@example.com"}}),
    event_processor.invoke({"user": {"email": "user@example.com"}})
)
```

```
True False
```

2.1.5 Bigger Apps & Modules

All these examples have assumed everything happens in a single file, which is not the case for most application. So if your application is a bit more substantial and you want to split it up into modules, this is how you do it.

Note: This example does not feature dependency factories, but it works the same way. You can simply add factories to subprocessors and they will automatically get added to the main processor when you call `add_subprocessor` on it. Also, if a factory with a given name already exists in the main processor, it will not be added again, so if you have multiple factories with the same name, but different behavior, the one in the main processor will be used.

Listing 1: my_processor.py

```python
from event_processor import EventProcessor


event_processor = EventProcessor()


@event_processor.processor({"key": "value"})
def my_processor(event):
    return event["key"]
```

Listing 2: main.py

```python
from event_processor import EventProcessor

from src.my_processor import event_processor as my_processor


main_processor = EventProcessor()
main_processor.add_subprocessor(my_processor)


def main(event):
    print(main_processor.invoke(event))
```

```
>>> main({"key": "value"})
value
```

The idea here is to define isolated processors in python submodules (as seen in `my_processor.py`) and to import those processors back into the main module, to add them as subprocessors to the main processor. You can add as many subprocessors as you want, but there can be no overlap in filter expressions, just like if you were always using the same event processor.

This is unfortunately required because of the way Python imports work. It would not be possible to instead import the main procesor from submodules, because then since nothing would import the submodules, the processors would never be registered.

2.2 Filtering Guide

Filters are required when using the `EventProcessor.processor()` decorator. They're used to determine which processor is responsible for processing which event.

2.2.1 General Usage

Filters are just python dictionaries that you need to pass to the decorator when decorating processors. Here's an example:

```
from typing import Any, Dict

from event_processor import EventProcessor, EventProcessorInvocationException


event_processor = EventProcessor()


@event_processor.processor({"top.mid.low": "nice", "top.other": Any})
def my_processor(event: Dict):
    return True

print(event_processor.invoke({"top": {"mid": {"low": "nice"}, "other": 1234}}))
```

```
True
```

You can specify any number of filters in the decorator, but there is an implicit AND between them, so ALL the filters need to match a particular event for your processor to be called.

2.2.2 Filter Keys

The key for a filter is just a string representation of a path inside the input event. This is best explained with an example, so imagine you have the following input event:

```
{"key": {"deeper": {"deepest": "something"}}}
```

The path to `something` is `key.deeper.deepest`. So, if you want to filter on the value associated with `deepest`, you can use this filter:

```
# Note that "interesting" should be the value that
# you expect to find at key.deeper.deepest.
{"key.deeper.deepest": "interesting"}
```

2.2.3 Filter Values

Once you can refer to a value in the event, it's useful to make assertions on what that value should be. If you don't care what the value is, but you just want the key to exist in the input dict, you can use the value `typing.Any` in your filter.

For example, the following filter will match any event that has a value at its specified key, regardless of what that value is:

```
from typing import Any

{"user.email": Any}
```

You might also want to filter on the actual value. In that case, just specify the value that should be present for your processor to be called like this:

```
{"user.role": "user"}
```

This filter could be used to only process events generated by users that have the "user" role, for example.

2.2.4 Match Everything

It can be really useful to have a default processor for any kind of event. For example, it can be used if an unexpected event is sent to be processed, and you don't want to miss events. To do this, simply use an empty filter. For example, the following filter will match any event:

```
{}
```

2.3 Pre-Processing Guide

Pre-processors are useful when you want to modify the input event before passing it on to your processor. It's mostly a convenience feature, because processors can always just accept the event and use it directly. Though, because pre-processors are any function, they can also fetch additional values not present in the event.

2.3.1 Structural Pre-Processing

One use of pre-processors is to change the structure of input events to make them more convenient to manipulate for processors. For example, you could turn an input event into a dataclass:

```
from dataclasses import dataclass
from typing import Any, Dict

from event_processor import EventProcessor


event_processor = EventProcessor()


@dataclass
class User:
    name: str
    email: str
```

(continues on next page)

(continued from previous page)

```
    role: str

def event_to_user(event: Dict) -> User:  # This is a pre-processor
    user = event["user"]
    return User(
        name=user["name"],
        email=user["email"],
        role=user["role"]
    )

@event_processor.processor(
    {"user.name": Any, "user.email": Any, "user.role": Any},
    pre_processor=event_to_user
)
def my_processor(user: User):  # The processor takes a User
    return user.role == "admin"

print(
    event_processor.invoke({"user": {"name": "John", "email": "john@example.com",
↪"role": "admin"}}),
    event_processor.invoke({"user": {"name": "Bob", "email": "bob@example.com", "role
↪": "user"}}),
)
```

```
True False
```

2.3.2 Data Pre-Processing

Another use of pre-processors is to fetch additional external data from, realistically, any source you could imagine. This can also be combined with dependencies to create dynamic pre-processors that can fetch data from external sources. Here's an example:

```
from event_processor import EventProcessor


event_processor = EventProcessor()


@dataclass
class User:
    name: str
    email: str
    role: str


class FakeDbClient:
    database = {
        "admin@example.com": {"role": "admin", "name": "John"},
        "user@example.com": {"role": "user", "name": "Bob"}
    }

    def fetch_by_email(self, email: str) -> User:
        user = self.database.get(email, {})
        return User(
```

(continues on next page)

(continued from previous page)

```python
            name=user["name"],
            email=email,
            role=user["role"]
        )


def event_to_user(event: Dict, db_client: FakeDbClient) -> User:
    email = event["user"]["email_3"]
    user = db_client.fetch_by_email(email=email)
    return user


@event_processor.dependency_factory
def db_client(_name: str) -> FakeDbClient:
    return FakeDbClient()


@event_processor.processor(
    {"user.email_3": Any},
    pre_processor=event_to_user,
    db_client=("my_db",)
)
def my_processor(user: User):
    return user.role == "admin"


print(
    event_processor.invoke({"user": {"email_3": "user@example.com"}}),
    event_processor.invoke({"user": {"email_3": "admin@example.com"}})
)
```

```
False True
```

For more details on dependency injection, see the *Dependency Injection Guide*. The gist is that you can specify dependencies in the decorator, and they will automatically be injected into either the processor, pre-processor, or both, depending on the parameters.

2.3.3 Bigger Data Pre-Processing Example

```python
from dataclasses import dataclass
from typing import Any, Dict

from event_processor import EventProcessor


event_processor = EventProcessor()


class FakeDynamoClient:
    database = {
        "users": [
            {"Email": {"S": "user@example.com"}, "Role": {"S": "user"}},
            {"Email": {"S": "admin@example.com"}, "Role": {"S": "admin"}}
        ]
```

(continues on next page)

(continued from previous page)

```
    }

    def get_item(self, TableName="", Key={}):
        table = self.database.get(TableName, {})
        key_name = list(Key.keys())[0]
        record = [e for e in table if e[key_name]["S"] == Key[key_name]["S"]][0]
        return {"Item": record}


@dataclass
class User:
    email: str
    role: str


@event_processor.dependency_factory
def boto_clients(client_name: str) -> FakeDynamoClient:
    if client_name == "dynamodb":
        return FakeDynamoClient()
    else:
        raise NotImplementedError()


# Uses the dynamodb client specified in the processor decorator
def event_to_user(event: Dict, dynamodb_client: FakeDynamoClient):
    email = event["user"]["email"]
    response = dynamodb_client.get_item(
                    TableName="users",
                    Key={"Email": {"S": email}}
               )
    role = response["Item"]["Role"]["S"]

    return User(email=email, role=role)


# Does not use the dynamodb client, but needs it for pre-processing
@event_processor.processor(
    {"user.email": Any},
    pre_processor=event_to_user,
    boto_clients=("dynamodb",)
)
def my_processor(user: User):
    return user.role == "admin"


print(
    event_processor.invoke({"user": {"email": "user@example.com"}}),
    event_processor.invoke({"user": {"email": "admin@example.com"}})
)
```

```
False True
```

2.4 Dependency Injection Guide

Sometimes, you want processors to take actions on external resources or services. To do this, it's usually necessary to use a client from a SDK, or a class you built yourself which encapsulates the state for your client.

These clients most often require authentication, API keys, or other such things. Dependency injection is a clean and convenient solution to the problem of supplying clients to event processors.

2.4.1 Overview

At its core, this library expects factory functions to exist for those clients (either made by you or not). These factories are used to create client instances, which will be forwarded to processors or pre-processors. You should use the `EventProcessor.dependency_factory()` decorator to register factory functions.

2.4.2 Registering Dependency Factories

Only factories that have been previously registered can be used in the processor decorator. Those factories should be registered with the `EventProcessor.dependency_factory()` decorator. Factories are just functions that take a single string argument (the client or SDK name) and return an instance of the client or SDK.

2.4.3 Combining Dependencies and Pre-Processors

This is a powerful use-case for both pre-processors and dependency injection. Since dependencies will be forwarded to the pre-processor (optionally) as well as the processor (also optionally), it's possible to ues pre-processors to make external API calls or to use a database. This keeps the processors very simple and it also allows the pre-processors to fully benefit from dependency injection. You can find an example of this use-case in the *Pre-Processing Guide*.

2.4.4 When does Injection Occur?

Forwarding occurs when dependencies are specified in the processor decorator and either the processor itself or the pre-processor require dependencies. They are determined to require dependencies whenever they take more than a single parameter.

For processors and pre-processors, the first parameter will always be the event (or the output of pre-processing), so other parameters will be dependencies. Do note that processors and pre-processors can either take no dependencies or all depencies, they cannot only take a few dependencies.

2.4.5 Full Example

An especially convenient use-case for dependency injection is the AWS boto3 client. The following example gives an idea for how you could use dependency injection to work with boto3 clients.

```python
from dataclasses import dataclass
from typing import Any, Dict

from event_processor import EventProcessor


event_processor = EventProcessor()

```

(continues on next page)

(continued from previous page)

```python
class FakeDynamoClient:
    database = {
        "users": [
            {"Email": {"S": "user@example.com"}, "Role": {"S": "user"}},
            {"Email": {"S": "admin@example.com"}, "Role": {"S": "admin"}}
        ]
    }

    def get_item(self, TableName="", Key={}):
        table = self.database.get(TableName, {})
        key_name = list(Key.keys())[0]
        record = [e for e in table if e[key_name]["S"] == Key[key_name]["S"]][0]
        return {"Item": record}


@dataclass
class User:
    email: str
    role: str


@event_processor.dependency_factory
def boto_clients(client_name: str) -> FakeDynamoClient:
    if client_name == "dynamodb":
        return FakeDynamoClient()
    else:
        raise NotImplementedError()


# Uses the dynamodb client specified in the processor decorator
def event_to_user(event: Dict, dynamodb_client: FakeDynamoClient):
    email = event["user"]["email"]
    response = dynamodb_client.get_item(
                    TableName="users",
                    Key={"Email": {"S": email}}
               )
    role = response["Item"]["Role"]["S"]

    return User(email=email, role=role)


# Does not use the dynamodb client, but needs it for pre-processing
@event_processor.processor(
    {"user.email": Any},
    pre_processor=event_to_user,
    boto_clients=("dynamodb",)
)
def my_processor(user: User):
    return user.role == "admin"


print(
    event_processor.invoke({"user": {"email": "user@example.com"}}),
    event_processor.invoke({"user": {"email": "admin@example.com"}})
)
```

```
False True
```

2.5 Testing Processors

Thanks to the separation between registering and invoking processors, testing every component of the system is extremely easy. Essentially, since the processor decorator does not modify the function it decorates, it's possible to test processors the same way any other function would be tested.

Because of the dependency injection, it's also very easy to mock clients during testing.

2.5.1 Example

Suppose that we have the following functions we want to test:

```
from dataclasses import dataclass
from typing import Any, Dict

from event_processor import EventProcessor


event_processor = EventProcessor()


class FakeDynamoClient:
    database = {
        "users": [
            {"Email": {"S": "user@example.com"}, "Role": {"S": "user"}},
            {"Email": {"S": "admin@example.com"}, "Role": {"S": "admin"}}
        ]
    }

    def get_item(self, TableName="", Key={}):
        table = self.database.get(TableName, {})
        key_name = list(Key.keys())[0]
        record = [e for e in table if e[key_name]["S"] == Key[key_name]["S"]][0]
        return {"Item": record}


@dataclass
class User:
    email: str
    role: str


@event_processor.dependency_factory
def boto_clients(client_name: str) -> FakeDynamoClient:
    if client_name == "dynamodb":
        return FakeDynamoClient()
    else:
        raise NotImplementedError()


# Uses the dynamodb client specified in the processor decorator
def event_to_user(event: Dict, dynamodb_client: FakeDynamoClient):
```

(continues on next page)

(continued from previous page)

```
    email = event["user"]["email"]
    response = dynamodb_client.get_item(
                    TableName="users",
                    Key={"Email": {"S": email}}
               )
    role = response["Item"]["Role"]["S"]

    return User(email=email, role=role)


# Does not use the dynamodb client, but needs it for pre-processing
@event_processor.processor(
    {"user.email": Any},
    pre_processor=event_to_user,
    boto_clients=("dynamodb",)
)
def my_processor(user: User):
    return user.role == "admin"


print(
    event_processor.invoke({"user": {"email": "user@example.com"}}),
    event_processor.invoke({"user": {"email": "admin@example.com"}})
)
```

```
False True
```

We could write the following tests:

```
from unittest.mock import MagicMock, patch


def test_my_processor_returns_true_for_admin_user():
    test_user = User(email="test@example.com", role="admin")

    result = my_processor(test_user)

    assert result is True


def test_event_to_user_returns_user_data_from_dynamodb():
    dynamodb_client = MagicMock()
    dynamodb_client.get_item.return_value = {
        "Item": {
            "Role": {"S": "mock-value"}
        }
    }
    test_event = {"user": {"email": "test@example.com"}}

    result = event_to_user(test_event, dynamodb_client)

    assert result.role == "mock-value"
    dynamodb_client.get_item.assert_called_once()


@patch(FakeDynamoClient)
```

(continues on next page)

(continued from previous page)

```python
def test_boto_clients_creates_boto_client(dynamo_client_mock):
    test_client_name = "mock-value"

    result = boto_clients(test_client_name)

    assert result == dynamo_client_mock.return_value
```

As you can see, the dependency injection makes the processor and pre-processor easy to test, and it makes those tests clearer by avoiding excessive patching. Patching *is* needed to test the dependency factory, but since that's the only thing to test, it doesn't make the test any less clear.

2.6 API Documentation

2.6.1 Event Processor

class `src.event_processor.event_processor.`**`EventProcessor`**

`add_subprocessor`(*subprocessor: src.event_processor.event_processor.EventProcessor*)

Add a subprocessor for events.

This will update the current event processor with all the processors of the subprocesor, which means that invoking the main processor with an event will have the same effect as invoking the correct subprocessor.

Note that filters defined in subprocessors must not already exist in the main processor, otherwise an error will be raised.

Parameters **`subprocessor`** – The subprocessor to add to the current processor.

Raises ***`EventProcessorSubprocessorException`*** – When there is an overlap in filter expressions between the processor and subprocessor.

`dependency_factory`(*fn: Callable*)

Register a dependency factory.

The name of the function will be the name of the factory, so this is what must be used in processor decorators. Also, the function must take a single string parameter and return a dependency based on that.

Parameters **`fn`** – The function that will act as a factory.

Raises

- ***`EventProcessorDecorationException`*** – When a factory is already registered for that name.
- ***`EventProcessorDecorationException`*** – When the decorated function does not have a single argument.

`invoke`(*event: Dict*) → Any

Invoke an event processor for the given event.

The correct processor will automatically be selected based on the event, and its dependencies will be automatically created and injected.

Parameters **`event`** – The raw event.

Returns The value returned by the processor.

Raises

- *EventProcessorInvocationException* – When no processor is found for the event.
- *EventProcessorDependencyException* – When a factory required by a processor was not registered.
- *EventProcessorDependencyException* – When the processor does not accept the right number of args.
- *EventProcessorDependencyException* – When the pre-processor does not accept the right number of args.

processor(*filter_expr: Dict[str, Any], pre_processor: Callable[[Dict], Any] = <function passthrough>, **kwargs*)

Decorate event processors.

Important Considerations

- The keyword arg in the decorator must match the dependency factory function's name.
- The arguments are passed in the following order : event, dependencies.
- All dependencies for a factory are passed before moving onto the next factory.
- The argument names are not important, but the order must be followed.

Parameters

- **filter_expr** – A dict containing path-value pairs to call the right event processor.
- **pre_processor** – A pre-processor function to transform the event into another type.
- **kwargs** – A mapping of dependency-factory to tuple of dependencies for that factory.

Raises *EventProcessorDecorationException* – When the filter expression is already associated to a handler.

2.6.2 Exceptions

Exceptions for event processor.

exception EventProcessorDecorationException(*msg: str, wrapped_fn: Callable*)

Exception for failures while wrapping processors.

exception EventProcessorDependencyException(*msg: str, wrapped_fn: Callable, dependencies: Dict[str, Tuple[str, ...]]*)

Exception for failures in dependencies.

exception EventProcessorException(*msg: str*)

General exception for the event-processor library.

exception EventProcessorInvocationException(*msg: str, event: Dict*)

Exception for failures in invocation.

exception EventProcessorSubprocessorException(*msg: str, overlap: Set*)

Exception for failures in subprocessor management.

2.6.3 Processor

Contains a class which represents an event processor.

class `processor.`**`Processor`**(*fn: Callable*, *pre_processor: Callable*, *dependencies: Dict[str, Tuple[str, . . .]]*)

Represent a registered processor.

2.6.4 Pre-Processors

Built-in event preprocessors.

`pre_processors.`**`passthrough`**(*event*)

Passthrough the event without touching it.

Parameters **`event`** – The input event.

Returns The input event.

CHAPTER THREE

CHANGELOG

- v1.1.0: Add support for subprocessors
- v1.0.0: Move the state and decorators inside a class
- v0.0.1: Initial release

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX