SIP-02 Python wrapper

Motivation

Current wrappers such as standalone (JVM) or distributed (Flink) already allow us to develop new processors in the given runtime environment. More and more people from the community ask to also support Python based processors. Especially, data scientists are likely to use this.

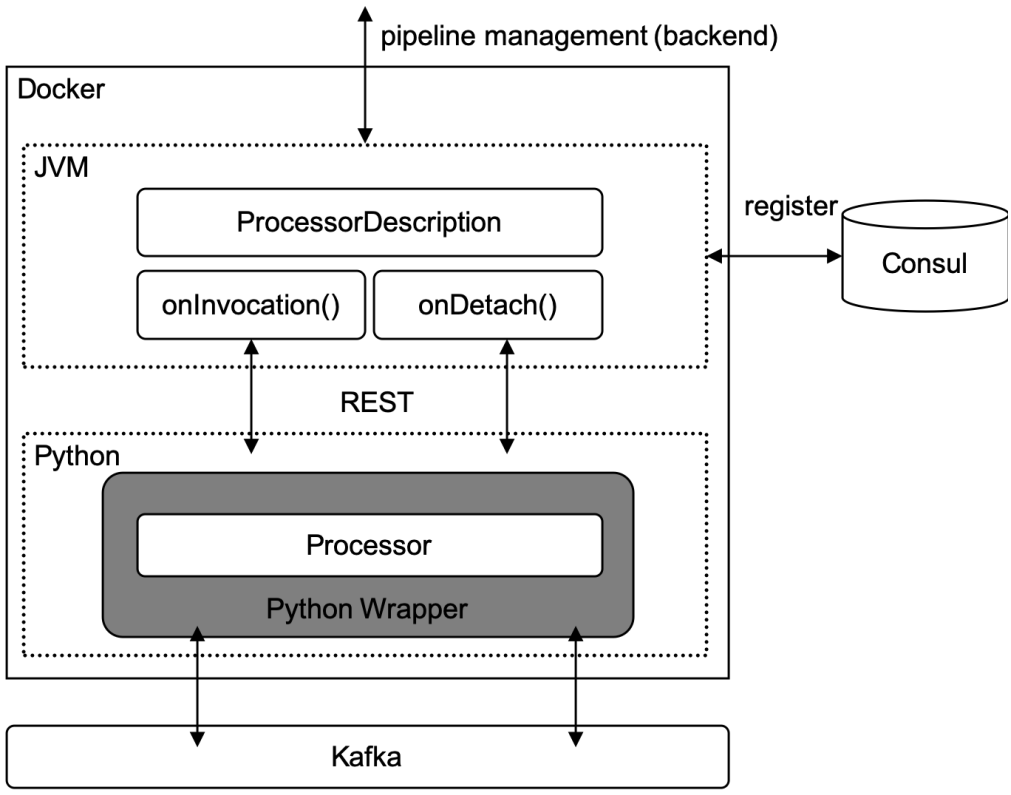
Why Python wrapper?

* Python is a widely used language especially in the domain of data science
* Python is more concise and thus better to read
* We provide more options for standalone algorithms: It allows newcomers unfamiliar with Java to faster implement their algorithms

POC

We implemented a rudimentary prototype that involves a hybrid solution using the Java side for model declaration and registration as well as the Python side, where the actual logic is implement. The two programs can be deployed via Docker using supervisord to start two services in a Docker container. Java talks to Python via REST while Python is exposing Flask endpoints for starting and stopping processor threads as depicted in the Figure below. The python part currently receives minimal information as JSON via REST that involves:

- **processor id**, e.g. org.apache.streampipes.examples.python.processor.greeter
- **invocation_id**, a UUID for keeping track of running instances of a given processor (one processor can potentially be used in multiple pipelines)
- **bootstrap_server**, the url to the kafka host, e.g. kafka:9092
- **input_topic**, the input topic to subscribe from
- **output_topic**, the dedicated output topic to produce to
- **static_properties**, the user defined static properties entered in the UI



The request is then posted to a flask endpoint exposed by the Python wrapper and uses the processor_id to start a new instance of the given processor, e. g. Greeter with the corresponding information from the request. Any processor extends an EventProcessor base class and implements 3 methods:

- **on_invocation()**, to get the static properties
- **on_event()**, where the actual application logic takes places and an arbitrary function is applied on the the input event
- **on_detach()**, to clean up (in case of any external connections established in the on_invocation method)

Example Greeter Processor

```
from streampipes.core import StandaloneSubmitter, EventProcessor
from streampipes.manager import Declarer


class Greeter(EventProcessor):
    greeting = None

    def on_invocation(self):
        # extract greeting text from static property
        self.greeting = self.static_properties.get('greeting')

    def on_event(self, event):
        event['greeting'] = self.greeting
        return event

    def on_detach(self):
        pass


def main():
    # dict with processor id and processor class
    processors = {
        'org.apache.streampipes.examples.python.processor': Greeter,
    }

    Declarer.add(processors=processors)
    StandaloneSubmitter.init()


if __name__ == '__main__':
    main()
```