PyPipeline Documentation

Release 1.0.0

Vaibhav Sinha

Aug 05, 2017

Contents

PyPipeline is an Enterprise Service Bus for Python. It is inspired from the Apache Camel project for Java. Like Camel, PyPipeline provides an intuitive Domain Specific Language to build a pipeline.

Here is an example which creates and runs a pipeline:

```
builder = DslPipelineBuilder()
pipeline = builder.source(Timer, {"period": 1.0}).to(Log, {"name": "test"}).build()
pipeline.start()
time.sleep(10)
pipeline.stop()
```

In the example, an instance of DslPipelineBuilder is created (This is the only PipelineBuilder implementation available right now. Later a YAML based PipelineBuilder might be added), which is used to specify the pipeline structure.

For any pipeline, a source is needed, which produces data that is to be sent down the pipe. In the example, the source is a Timer which is configured to produce data every 1 second. The data is then sent to a Log Destination, which prints the data to standard out.

The pipeline is then started, which triggers the Timer to start sending data. After 10 seconds, the pipeline is stopped.

CHAPTER 1

Enterprise Integration Patterns

PyPipeline intends to support all of patterns from the Enterprise Integration Patterns book by Gregor Hohpe and Booby Woolf.

Currently the following patterns are supported

- Filter
- Aggregator
- Splitter
- Multicast
- Content Based Router
- Routing Slip
- Dynamic Router
- Resequencer
- Validator
- Wiretap

Following EIPs can be implemented using the patterns listed above:

- Recipient List
- Composed Message Processor
- Scatter Gather
- Content Enricher
- Content Filter
- Claim Check
- Normalizer
- Detour

- Log

CHAPTER 2

Concepts:

PyPipeline Core

PyPipeline has five major concepts

1. Components/Endpoints
2. Exchange
3. Pipeline
4. Enterprise Integration Patterns
5. Plumber

Components/Endpoints

Components are the blocks which either produce, process or consume data that flows on the bus.

In PyPipeline, the components can be of two types:

Source

A source is a component which produces data that will flow on the bus. How the data is generated is completely up to the source. Few examples of how sources may generate data are as follows

- A MySQL source component may read a table from a database and send each row as one data packet.
- A RabbitMQ source may listen to a queue and produce a data packet for every message received from the queue.
- A REST source may access an API and send the response as a data packet

The sources continuously run, until explicitly stopped, and are event based. They run their own thread, which responds to some event (a message on the queue, in case of RabbitMQ source) and produce data packets.

A source might take some parameters for configuration, such as database name, host and port in case of MySQL source.

When implementing any source, the class must extend the Source class from *core* package.:

```
class Source:
    def __init__(self, plumber, params):
        self.plumber = plumber
        self.chain = None
        self.params = params

    def start(self):
        raise NotImplementedError("Sources should implement their start method")

    def stop(self):
        raise NotImplementedError("Sources should implement their stop method")
```

When start is called, the source must start it's own thread which listens to the events and produces data packets. To send the data of the bus, it must call the *process* method of *self.chain*.

When stop is called, the source must stop the thread and not send any more data packets on the bug.

Destination

A destination is a component which receives a data packet as input and processes it to either modify the data, or send it an external service, or both. Few examples of destination are as follows

- A MongoDB destination component which receives json data and persists it in a collection
- A reverse geocoder destination which modifies the data to convert it location field from lat/long to address
- A log destination component which prints the content of the data on console

A destination may take some parameters for configuration, such as collection name, host and port in case of MongoDB destination.

When implementing any destination, the class must extend the Destination class from *core* package.:

```
class Destination:
    def __init__(self, plumber, params):
        self.plumber = plumber
        self.params = params

    def process(self, exchange):
        raise NotImplementedError("Subclass of destination needs to implement process␣
↪method")
```

The destination class should implement it's logic in the *process* method.

Exchange

The data packets that flow on the bus are encapsulated with an Exchange.:

```
import uuid


class Exchange:
    def __init__(self):
```

```
        self.id = uuid.uuid4()
        self.in_msg = None
        self.out_msg = None
        self.properties = {}

    def __str__(self):
        return "Id: " + str(self.id) + "\nProperties: " + str(self.properties) +
↪"\nIn Msg:\n" + str(self.in_msg) + "\nOut Msg:\n" + str(self.out_msg)
```

The exchange has an id, a properties dict, and in and out messages. The in and out messages are of type Message.:

```
class Message:
    def __init__(self):
        self.headers = {}
        self.body = None

    def __str__(self):
        return "\tHeaders: " + str(self.headers) + "\n\tBody: " + str(self.body)
```

Each message has a headers dict and a body. The body may contain any data.The actual data that the source wants to send is put within the body.

The *in_msg* in Exchange is generally used by most components. The *out_msg* is needed only for request-reply EIP.

Pipeline

The pipeline represents the bus. It is a sequence of components, starting with a Source and followed by arbitrary number of Destination. The data travels on the bus, according to Pipeline configuration.

If I wanted to take data from a Timer based source, modify it in a MessageModifier destination, and then send it to a Log destination, then the pipeline would be built as follows:

```
class PipelineTest(unittest.TestCase):

    def test_simple_pipeline(self):
        builder = DslPipelineBuilder()
        pipeline = builder.id("pipeline1").source(Timer, {"period": 1.0}).
↪to(MessageModifier).to(Log, {"name": "test"}).build()
        pipeline.start()
        time.sleep(10)
        pipeline.stop()


class MessageModifier(Destination):
    def process(self, exchange):
        exchange.in_msg.body += " modified"
```

A pipeline must have one and only one source.

A pipeline id can also be specified, using which the pipeline can be referenced. More on that later.

Enterprise Integration Patterns

PyPipeline implements many Enterprise Integration Patterns. A pattern might have its own method in the DslPipelineBuilder or it might be possible to implement the pattern by combining other patterns.

One of the EIP is Message Filter. As evident by the name, it is a step in the pipeline which filters messages based on some criteria that is provided as configuration. Example of Filter is as follows:

```
class FilterTest(unittest.TestCase):

    def test_simple_pipeline(self):
        builder = DslPipelineBuilder()
        pipeline = builder.source(Timer, {"period": 1.0}).filter(filter_method).
↪process(Log, {"name": "test"}).build()
        pipeline.start()
        time.sleep(10)
        pipeline.stop()


def filter_method(exchange):
    parts = exchange.in_msg.body.split()
    return int(parts[-1]) % 2 == 0
```

In this example, the filter() step takes a method which it will use to determine which messages should be allowed to continue on the pipeline.

Plumber

Plumber is the Pipeline manager. It can be used to register multiple PipelineBuilders. It then builds the pipelines. All the pipelines can then be started by calling start() on the plumber. Similarly, all the pipelines can be stopped by calling stop on the plumber. With the use of plumber, it is also possible to start/stop individual pipelines by calling start_pipeline/stop_pipeline and providing the pipeline id.

When a pipeline is built by registering the builder with the Plumber, then the pipeline and all its components are provided with the instance of the plumber. This can be used to do advanced stuff from within a component, such as starting another pipeline which is registered with the Plumber.

The plumber usage is as follows:

```
class PlumberTest(unittest.TestCase):

    def test_simple_pipeline(self):
        plumber = Plumber()
        builder1 = DslPipelineBuilder()
        builder2 = DslPipelineBuilder()
        pipeline1 = builder1.source(Timer, {"period": 1.0}).to(MessageModifier)
        pipeline2 = builder2.source(Timer, {"period": 2.0}).to(MessageModifier)
        plumber.add_pipeline(pipeline1)
        plumber.add_pipeline(pipeline2)
        plumber.start()
        time.sleep(10)
        plumber.stop()


class MessageModifier(Destination):
    def process(self, exchange):
        exchange.in_msg.body += " modified"
```