PyPipeline – интеграционный фреймворк для Python

Основные концепции:

- **Компоненты** / Конечные точки обмена (Endpoints)
- **Сущность «Обмен»** (Exchange)
- **«Трубопровод»** (конвейер) преобразований (Pipeline)
- **EIP**
- **Менеджер** «трубы» (Plumber)

Компоненты

Компоненты - это блоки, которые либо производят, обрабатывают или потребляют данные, которые передаются по шине.

В PyPipeline компоненты могут быть двух типов:

- Источник (Source)

- Приёмник (Destination)

Источник

Это компонент, создающий данные, которые будут передаваться по шине. Как данные генерируются, полностью зависит от источника. Вот несколько примеров того, как источники могут генерировать данные:

- Исходный компонент **MySQL** может считывать таблицу из базы данных и отправлять каждую строку как один пакет данных.
- Источник **RabbitMQ** может прослушивать очередь и создавать пакет данных для каждого сообщения, полученного из очереди.
- Источник **REST** может получить доступ к API и отправить ответ в виде пакета данных.

Источники непрерывно работают, пока не будут явно остановлены, и основаны на событиях. Они запускают свой собственный поток, который отвечает на какое-то событие (сообщение в очереди, в случае источника **RabbitMQ**) и создает пакеты данных.

Источник может принимать некоторые параметры для конфигурации, такие как имя базы данных, хост и порт в случае источника MySQL.

При реализации любого источника класс должен расширять класс ***Source*** из пакета ***core***:

```
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")
```

Когда вызывается ***start***, источник должен запустить собственный поток, который прослушивает события и генерирует пакеты данных. Чтобы отправить данные в шину, необходимо вызвать метод ***process*** от ***self.chain***.

Когда вызывается ***stop***, источник должен остановить поток и не отправлять больше пакетов данных в шину.

Приёмник

Пункт назначения или Приёмник - это компонент, который принимает пакет данных в качестве входных данных и обрабатывает его, чтобы либо изменить данные, либо отправить его внешней службе, либо и то, и другое. Вот несколько примеров назначения:

- Компонент назначения ***MongoDB***, который получает данные json и сохраняет их в коллекции
- Обратный пункт назначения геокодера, который изменяет данные, чтобы преобразовать их поле местоположения из широты / долготы в адрес
- Компонент назначения журнала (логгера), который печатает содержимое данных на консоли

Приёмник может принимать некоторые параметры для конфигурации, такие как ***имя коллекции***, ***хост*** и ***порт*** в случае назначения ***MongoDB***.

При реализации любого пункта назначения класс должен расширять класс ***Destination*** из пакета ***core***:

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

    def process(self, exchange):
        raise NotImplementedError("destination must implement process method")
```

Класс ***Приёмник*** должен реализовывать свою логику в методе ***process***.

Обмен (Exchange)

Пакеты данных, которые передаются по шине, инкапсулированы в 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)
```

Сущность «Обмен» имеет идентификатор (***id***), словарь свойств, и входящее (***in***) и исходящее (***out***) сообщения. Входящее и исходящее сообщения имеют тип ***Message***:

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

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

Каждое сообщение имеет словарь заголовков (***headers***) и тело (***body***). Тело может содержать любые данные. Фактические данные, которые хочет отправить источник, помещаются в тело.

«Трубопровод»

«Трубопровод» представляет собой собственно шину интеграции. Это последовательность компонентов, начиная с источника и заканчивая произвольным количеством Пунктов назначения. Данные передаются по шине в соответствии с конфигурацией конвейера.

Если, например, взять данные из источника на основе таймера (***Timer***), изменить их в ***MessageModifier - приёмнике***, а затем отправить в место назначения Журнал (***Log***), тогда конвейер будет выглядеть следующим образом:

```
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"
```

«Трубопровод» должен иметь один и только один источник.

Также можно указать идентификатор конвейера, используя который можно ссылаться на конвейер. Подробнее об этом позже.

Шаблоны корпоративной интеграции (EIP)

PyPipeline реализует множество корпоративных шаблонов интеграции. Шаблон может иметь свой собственный метод в ***DslPipelineBuilder***, или может быть возможно реализовать шаблон путем объединения других шаблонов.

Одним из ***EIP*** является ***Фильтр*** сообщений. Как видно из названия, это шаг в конвейере, который фильтрует сообщения на основе некоторых критериев, предоставляемых в качестве конфигурации. Пример фильтра выглядит следующим образом:

```
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
```

В этом примере шаг «трубы» ***filter()*** использует метод, с помощью которого он будет определять, какие сообщения должны быть оставлены в конвейере, а какие нет.

Менеджер «трубы» (*Plumber)*

Менеджер «трубы может быть использован для регистрации нескольких ***PipelineBuilders***. После чего все эти конвейеры можно запустить, вызвав ***start()*** для ***менеджера***.

Точно так же все «трубопроводы» можно остановить, вызвав команду ***stop*** на менеджере. С использованием менеджера «трубы» также можно запускать / останавливать отдельные конвейеры, вызывая start_pipeline / stop_pipeline и предоставляя идентификатор конвейера, сосредоточив таким образом, управление в единой точке.

Когда конвейер строится путем регистрации ***builder*** в менеджере «трубы», тогда этому конвейеру и всем его компонентам предоставляется экземпляр менеджера. Это можно использовать для выполнения сложных действий внутри компонента, таких как запуск другого конвейера, который зарегистрирован в менеджере (***Plumber)***.

Использование менеджера заключается в следующем:

```
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"
```

Пример простейшего приложения

```
#!/usr/bin/env python

import time
from pypipeline.components.source.Timer import Timer
from pypipeline.core.DslPipelineBuilder import DslPipelineBuilder
from pypipeline.core.Plumber import Plumber

class Filter:
    def __call__(self, exchange):
        parts = exchange.in_msg.body.split()
        return int(parts[-1]) % 2 == 0

def main():
        plumber = Plumber()
        builder1 = DslPipelineBuilder()

        pipeline1 = builder1 \
        .source(Timer, {"period": 1.0}) \
        .filter(Filter()) \
        .process(lambda ex: print(ex.in_msg.body))

        plumber.add_pipeline(pipeline1)
        plumber.start()
        time.sleep(10)
        plumber.stop()


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


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

=== Результаты запуска: ===
This is exchange 0
This is exchange 2
This is exchange 4
This is exchange 6
This is exchange 8
===