BSPump Reference Documentation Documentation

Release v1903

TeskaLabs

Jun 06, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

BitSwan is a product designed to real-time data processing. By means of so-called real-time processors BitSwan is able to analyze hundreds of data streams from a lot of various sources at the same time, which makes it suitable to detect anomalies and data patterns as well as other situations when instantaneous action is needed. BitSwan is based on Python language.

CHAPTER

TWO

HOW TO INSTALL BITSWAN

Use command in your command prompt

```
pip install bspump
```

or you can clone the github repository BitsSwanPump

```
pip install git+https://github.com/LibertyAces/BitSwanPump.git
```

2.1 How it works

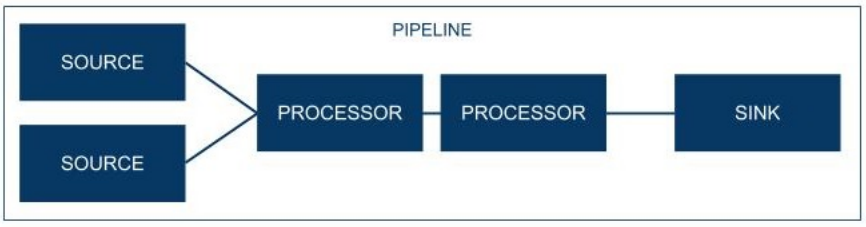
heeelp

2.1.1 Pipeline

Pipeline is responsible for **data processing** in BSPump. Individual *Pipeline* objects work **asynchronously** and **independently** of one another (provided dependence is not defined explicitly – for instance on a message source from some other pipeline). Each *Pipeline* is usually in charge of **one** concrete task.

Pipeline has three main components:

- *Source*
- *Processor*
- *Sink*



BitSwan PIPELINE

Source connects different **data sources** with the *Pipeline* to be processed

Multiple sources

A *Pipeline* can have multiple sources. They are simply passed as a list of sources to a *Pipeline build()* method.

```
class MyPipeline(bspump.Pipeline):

   def __init__(self, app, pipeline_id):
      super().__init__(app, pipeline_id)
      self.build(
         [
            MySource1(app, self),
            MySource2(app, self),
            MySource3(app, self),
         ]
         bspump.common.NullSink(app, self),
      )
:meta private:
```

The main component of the BSPump architecture is a so-called *Processor*. This object **modifies**, **transforms** and **enriches** events. Moreover, it is capable of **calculating metrics** and **creating aggregations**, **detecting anomalies** or react to known as well as unknown **system behaviour patterns**.

Processors differ as to their **functions** and all of them are aligned according to a predefined sequence in **pipeline objects**. As regards working with data events, each *Pipeline* has its unique task.

Processors are passed as a **list** of *Processors* to a *Pipeline build()* method

```
class MyPipeline(bspump.Pipeline):

   def __init__(self, app, pipeline_id):
      super().__init__(app, pipeline_id)
      self.build(
         [
            MyProcessor1(app, self),
            MyProcessor2(app, self),
            MyProcessor3(app, self),
         ]
         bspump.common.NullSink(app, self),
      )
:meta private:
```

Sink object serves as a **final event destination** within the pipeline given. Subsequently, the event is dispatched/written into the system by the BSPump

2.1.2 Source

Source is an **object** designed to obtain data from a predefined input. The BSPump contains a lot of universally usable, specific source objects, which are capable of loading data from known data interfaces. The BitSwan product further expands these objects by adding source objects directly usable for specific cases of use in industry field given.

Each source represent a coroutine/Future/Task that is running in the context of the main loop. The coroutine method *main()* contains an implementation of each particular source.

Source MUST await a *Pipeline* ready state prior producing the event. It is acomplished by *await self.Pipeline.ready()* call.

Trigger Source

This is an abstract source class intended as a base for implementation of 'cyclic' sources such as file readers, SQL extractors etc. You need to provide a trigger class and implement *cycle()* method.

Trigger source will stop execution, when a *Pipeline* is cancelled (raises concurrent.futures.CancelledError). This typically happens when a program wants to quit in reaction to a on the signal.

You also may overload the *main()* method to provide additional parameters for a *cycle()* method.

```
async def main(self):
        async with aiohttp.ClientSession(loop=self.Loop) as session:
                await super().main(session)


async def cycle(self, session):
        session.get(...)
```

2.1.3 Processor

The main component of the BSPump architecture is a so called *processor*. This object modifies, transforms and enriches events. Moreover, it is capable of calculating metrics and creating aggregations, detecting anomalies or react to known as well as unknown system behavior patterns.

Processors differ as to their functions and all of them are aligned according to a predefined sequence in pipeline objects. As regards working with data events, each pipeline has its own unique task.

2.2 Bitswan Tutorials

2.2.1 Bitswan Tutorials

in this series of tutorials we will walk you through basic and more advanced examples and demos to initiate your adventure with BSPump.

You will learn more about the BSPump architecture and how each component works. However, before you can start on your journey you should know basics of python and be able to set up your programming environment.

Prerequisites

Here are some quick tutorials that will help you installing python and BSPump module using package installer for Python called pip.

Installing python

Firstly you should check whether you don't already have python installed. Open your command line or terminal and type:

```
C:/> python --version
> Python 3.8.4
```

if your python version is lower than 3.8 check Python.org

If you are a complete beginner to python or you want more information about python check out the Python tutorial

Installing BSPump module

To install BSPump module:

```
pip install asab bspump
```

or alternatively using

```
pip install git+https://github.com/LibertyAces/BitSwanPump-BlankApp.git
```

If you dont have installed pip type:

```
python get-pip.py
```

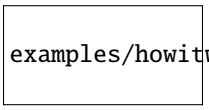
To check the version use.

```
pip --version
```

Have you managed to install everything? Then you are ready for creating your first BSPump.

BSPump Highlevel architecture

BSPump is made from several components which are going to be explained in this tutorial. As you probably know, Bitswan is a real-time stream processor. To be able to process and work with large amount of data, BSpump uses so-called Event Stream Processing, data is propagated through a data pipeline in Events. Event is a single data point with a timestamp. To handle these events Pipeline has special components that be compatible with each other. .Therefore, each pipeline is made from several vital compoents: source, processor and sink. However, for the pipeline to work Bitswan uses BSPump Service to handle and register connetions, pipelines etc.



Firstly, we will walk you through each of components and its functionality, so you can later build your own pipeline. Doesn't that sounds cool?

BSpump Service

Service is part where pipelines and connections are registered.

We will go through the following code and explain each part

```
import asab

from .pipeline import TCPPipeline


class BlankService(asab.Service):

    def __init__(self, app, service_name="blank.BlankService"):
        super().__init__(app, service_name)

    async def initialize(self, app):
        svc = app.get_service("bspump.PumpService")

        # Create and register all connections here

        # Create and register all matrices here

        # Create and register all lookups here

        # Create and register all pipelines here

        self.TCPPipeline = TCPPipeline(app, "TCPPipeline")
        svc.add_pipeline(self.TCPPipeline)

        await svc.initialize(app)

    async def get_data(self, message="be"):
        await self.TCPPipeline.process(message)
        return "Check stdout"
```

In this example we

Connection

To be able to connect to a data source you have to make a connection. connection is usually done in Source class and then registered in service class.

Pipeline

pipeline

```
import sys

import bspump
import bspump.common
import bspump.socket

```

(continues on next page)

(continued from previous page)

```
from .processor import ShakespeareanEnricher


class TCPPipeline(bspump.Pipeline):
    """
    To test this pipeline, use:
    socat STDIO TCP:127.0.0.1:8888
    or visit http://localhost:8080/blank?message=die
    """

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            bspump.socket.TCPSource(app, self, config={"host": "0.0.0.0", "port": 8888}),
            ShakespeareanEnricher(app, self),
            bspump.common.PPrintSink(app, self, stream=sys.stderr)
        )
```

Lookup

Source

Description about source. What is it ..

Streaming Source

Streaming Source enables events to enter in so-called stream. Events flow through source in real time manner as they are being delivered by the input technology.

Following technologies can be used as a streaming source

1. Kafka
2. Elastic Search
3. RabbitMQ

Elastic Search Source

TODO

Description

Example

Explanation

Kafka Source

TODO

Description

Example

Explanation

Trigger Source

Unlike streaming source, Trigger Source is used when we need to pump data from SQL-like databases or files. They have to be triggered by an external event or a repeating timer (requesting JSON data from APIs every 10 minutes).

Trigger Source can be used for:

1. HTTP client/server
2. SQL query
3. TCP
4. Files: csv, json etc.

TCP source

Description

TCP Source can be to obtain data from peer to peer connection using TCP.

Use case

TODO

Example

```
class EchoPipeline(bspump.Pipeline):

    '''
    To test this pipeline, use:
    socat STDIO TCP:127.0.0.1:8083
    '''

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)
        self.build(
            bspump.ipc.StreamServerSource(app, self, config={'address': '0.0.0.0 8083'}),
        )
```

HTTP Client Source

Description

HTTP Client Source gets data from a specified API URL.

Use case

if you need pump data from a single API URL you can use this Source.

Example

```
class SamplePipeline(bspump.Pipeline):

def __init__(self, app, pipeline_id):
    super().__init__(app, pipeline_id)

    self.build(
        bspump.http.HTTPClientSource(app, self, config={
            'url': '<<API URL>>'
        }).on(<<Here you will use some type of trigger>>),
    )
```

The API URL can be any API you wish to get data from.

You will need to specify your Trigger type. You can choose your Trigger here : TODO <<reference>>

Note

Full functional example with this source can be found here coindesk

MySQL

Description

Example

Explanation

JSON File

Description

Example

Explanation

CSV File

Description

Example

Explanation

Processor

Processor

```
import bspump


class ShakespeareanEnricher(bspump.Processor):

    def process(self, context, event):
        if isinstance(event, bytes):
            event = event.decode("utf-8").replace('\r', '').replace('\n', '')
        return 'To {0}, or not to {0}?'.format(event)
```

Sink

Sink is the part responsible for the output of the data to a database, standard output in your computer on into another pipeline.

PPrintSink

In this example we are going to use PPrintSink which prints the data from pipeline to stdout or any other stream that is connected to the pipeline.

To use sink in your pipeline

```
self.build(
    bspump.common.PPrintSink(app, self, stream=sys.stderr)
)
```

PPrintSink class is added to your pipeline. It should be the last part of the pipeline for the pipeline to work correctly.

to further explain the , *bspump.common.* is the part where you specify the path to the class *PPrintSink* is the name of the class. In the parentheses you can specify the output stream. If none is specified stdout is used.

code

```
class PPrintSink(Sink):
    """
    Description:

    |

    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, app, pipeline, id=None, config=None, stream=None):
        """
        Description:

        |

        """
        super().__init__(app, pipeline, id, config)
        self.Stream = stream if stream is not None else sys.stdout
```

The whole code can be found at BitSwan BlankApp

2.2.2 Coindesk API Example

About

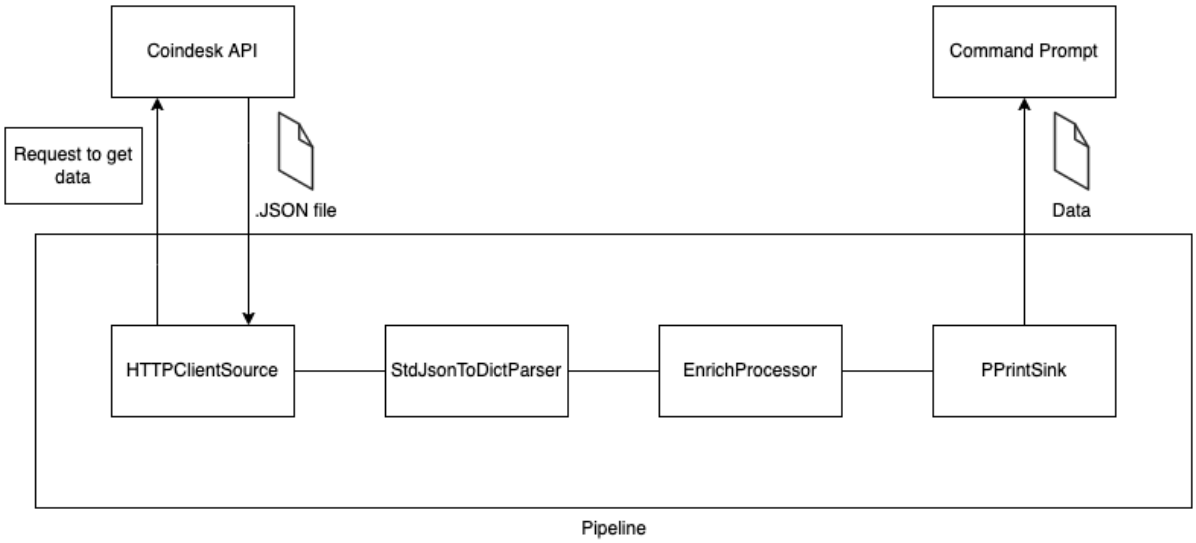
In this example we will learn how to extract any data from API. We will be using a HTTP Client Source for the API request.

In this example we will be using API from Coindesk to get the current price of Bitcoin.

The final pipeline will simply get data from the API request as a JSON, covert it to python dictionary, and output the data to Command Prompt. Additionally, I will show you how to create your own Processor to enrich the data.

The following code can be found here in our GitHub repo.

A diagram of the final pipeline.



Source and Sink

In the code below, you can see the basic structure of a pipeline. The important part is the `self.build()` method, where its parameters are the single components of the pipeline. In this part we will use two main components each pipeline must contain: Source and Sink. Do not copy this part of code yet, because it is not example on its own

```
class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            #Source of the pipeline
            bspump.http.HTTPClientSource(app, self, config={
                'url': 'https://api.coindesk.com/v1/bpi/currentprice.json'
            }).on(bspump.trigger.PeriodicTrigger(app, 5)),
            #Sink of the pipeline
            bspump.common.PPrintSink(app, self),
        )
```

Source is a component that supplies the pipeline with data. In our example we will use a specific type of Source. Because we need to Pump data from API, we need to send a request to the API to receive our data. This means that our Source has to regularly and send the request using API. For this reason we will be using so-called Trigger Source. More about *Trigger Source* .

HTTP Client Source can have many configurations, but in our example we just need to specify our URL address, using `config={'url': '<OUR URL>'}` as parameter in HTTP Client Source.

Because we are using Trigger Source, we need to specify which Trigger we will be using. There are many types of Triggers, but in our example we will be using PeriodicTrigger, which triggers in time intervals specified in the parameter. `bspump.trigger.PeriodicTrigger(app, <<Time parameter in seconds>>))`

Each pipeline has to have Sink. In our example we want to see the result of the data, so we will be using PPrintSink, which simply prints the data to the Command Prompt.

You can try to copy paste this chunk of code and try it yourself. Make use you have BSPump module installed for your Python, if not you can follow our guide *Installing BSPump module* .

```
#!/usr/bin/env python3
import bspump
import bspump.common
import bspump.http
import bspump.trigger

class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            bspump.http.HTTPClientSource(app, self, config={
                'url': 'https://api.coindesk.com/v1/bpi/currentprice.json'
            }).on(bspump.trigger.PeriodicTrigger(app, 5)),
            bspump.common.PPrintSink(app, self),
        )
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    app = bspump.BSPumpApplication()
    svc = app.get_service("bspump.PumpService")
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)
    app.run()
```

The program should output a JSON similar to this

```
(b'{"time":{"updated":"Jan 31, 2022 15:47:00 UTC","updatedISO":"2022-01-31T15:4'
 b'7:00+00:00","updateduk":"Jan 31, 2022 at 15:47 GMT"},"disclaimer":"This data'
 b' was produced from the CoinDesk Bitcoin Price Index (USD). Non-USD currency '
 b'data converted using hourly conversion rate from openexchangerates.org","cha'
 b'rtName":"Bitcoin","bpi":{"USD":{"code":"USD","symbol":"&#36;","rate":"37,789'
 b'.6250","description":"United States Dollar","rate_float":37789.625},"GBP":{"'
 b'code":"GBP","symbol":"&pound;","rate":"28,145.2970","description":"British P'
 b'ound Sterling","rate_float":28145.297},"EUR":{"code":"EUR","symbol":"&euro;"'
 b',"rate":"33,772.9280","description":"Euro","rate_float":33772.928}}}')
```

As you can see this is not ideal format to read our data from. We will need to convert our incoming data.

Your First Processor

After we have a functional pipeline, we can start with the more interesting part, Processors. The Processor is the component which works with data of an event. In this example we will use a simple Processor, StdJsonToDictParser, which only converts the incoming JSON to python Dict type, that is much easier to work with in python.

```
class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            bspump.http.HTTPClientSource(app, self, config={
                'url': 'https://api.coindesk.com/v1/bpi/currentprice.json'
            }).on(bspump.trigger.PeriodicTrigger(app, 5)),
            bspump.common.StdJsonToDictParser(app, self),
            bspump.common.PPrintSink(app, self),
        )
```

this Processor is added simply by adding it to `self.build()` between Source and Sink.

You should be getting more organized output

```
{'bpi': {'EUR': {'code': 'EUR',
                 'description': 'Euro',
                 'rate': '33,794.5989',
                 'rate_float': 33794.5989,
                 'symbol': '&euro;'},
         'GBP': {'code': 'GBP',
                 'description': 'British Pound Sterling',
```

(continues on next page)

(continued from previous page)

```
                'rate': '28,163.3569',
                'rate_float': 28163.3569,
                'symbol': '&pound;'},
        'USD': {'code': 'USD',
                'description': 'United States Dollar',
                'rate': '37,813.8733',
                'rate_float': 37813.8733,
                'symbol': '&#36;'}},
 'chartName': 'Bitcoin',
 'disclaimer': 'This data was produced from the CoinDesk Bitcoin Price Index '
               '(USD). Non-USD currency data converted using hourly conversion '
               'rate from openexchangerates.org',
 'time': {'updated': 'Jan 31, 2022 15:49:00 UTC',
          'updatedISO': '2022-01-31T15:49:00+00:00',
          'updateduk': 'Jan 31, 2022 at 15:49 GMT'}}
```

Creating Custom Processor

Because a most of your use cases will be unique, it is most likely that there will be no existing Processor that could do the work. Consequently, you will have to implement your own Processor.

Creating new Processor is not a complicated task. You will need to follow the basic structure of an general Processor. You can simply copy-paste the code below:

```
class EnrichProcessor(bspump.Processor):
   def __init__(self, app, pipeline, id=None, config=None):
      super().__init__(app, pipeline, id=None, config=None)

   def process(self, context, event):

      return event
```

This a sample processor class. The most important part of this processor class is the process method. This method will be called when an event is passed to the Processor. As you can see, the default implementation of process method returns the event *return event*. Event must be passed to the following component, another Processor or Sink.

If you wish to use your new Processor in our case *EnrichProcessor* You will need to reference it in *self.build* method. You can do that simply by adding it to *self.build* parameters.

```
class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            bspump.http.HTTPClientSource(app, self, config={
                'url': 'https://api.coindesk.com/v1/bpi/currentprice.json'
            }).on(bspump.trigger.PeriodicTrigger(app, 5)),
            bspump.common.StdJsonToDictParser(app, self),
            EnrichProcessor(app, self),
            bspump.common.PPrintSink(app, self),
        )
```

The last step is implementation. For our example, I created a simple script that takes the incoming event (python dictionary that contains price of Bitcoin in USD, Euro, and Pounds) and adds a new branch with a Japanese yen. The EnrichProcessor class has a new method *convertUSDtoJPY* which calculates the price of yen based on USD conversion rate (Note: The exchange rate is outdated for sake of simplicity of this example).

```
class EnrichProcessor(bspump.Processor):
 def __init__(self, app, pipeline, id=None, config=None):
     super().__init__(app, pipeline, id=None, config=None)

 def convertUSDtoJPY(self, usd):
     return usd * 113.70 #outdated rate usd/jpy

 def process(self, context, event):
     jpyPrice = str(self.convertUSDtoJPY(event["bpi"]["USD"]["rate_float"]))

     event["bpi"]["JPY"] = {
         "code": "JPY",
         "symbol": "&yen;",
         "rate": ''.join((jpyPrice[:3], ',', jpyPrice[3:])),
         "description": "JPY",
         "rate_float": jpyPrice
     }

     return event
```

When we add all parts together we get this functional code.

Your ouput should look something like this:

```
{'bpi': {'EUR': {'code': 'EUR',
                 'description': 'Euro',
                 'rate': '33,796.7930',
                 'rate_float': 33796.793,
                 'symbol': '&euro;'},
         'GBP': {'code': 'GBP',
                 'description': 'British Pound Sterling',
                 'rate': '28,165.1854',
                 'rate_float': 28165.1854,
                 'symbol': '&pound;'},
         'JPY': {'code': 'JPY',
                 'description': 'JPY',
                 'rate': '429,9716.52771',
                 'rate_float': '4299716.52771',
                 'symbol': '&yen;'},
         'USD': {'code': 'USD',
                 'description': 'United States Dollar',
                 'rate': '37,816.3283',
                 'rate_float': 37816.3283,
                 'symbol': '&#36;'}},
 'chartName': 'Bitcoin',
 'disclaimer': 'This data was produced from the CoinDesk Bitcoin Price Index '
               '(USD). Non-USD currency data converted using hourly conversion '
               'rate from openexchangerates.org',
 'time': {'updated': 'Jan 31, 2022 15:53:00 UTC',
```

(continues on next page)

(continued from previous page)

```
        'updatedISO': '2022-01-31T15:53:00+00:00',
        'updateduk': 'Jan 31, 2022 at 15:53 GMT'}}
```

To Summarize what we did in this example:

1. we created a sample pipeline with a Source and Sink
2. we added a new Processor that converts incoming events to python dictionary
3. we created a custom Processor that adds a information about Japanese currency to the incoming event and passes it to Sink .

Next steps

You can change and modify the pipeline in any manner you want. For example, instead of using PPrintSink you can use our Elasticsearch Sink which loads the data to Elasticsearch. Read more about *How to connect to Elastic Search* .

2.2.3 Weather API Example

About

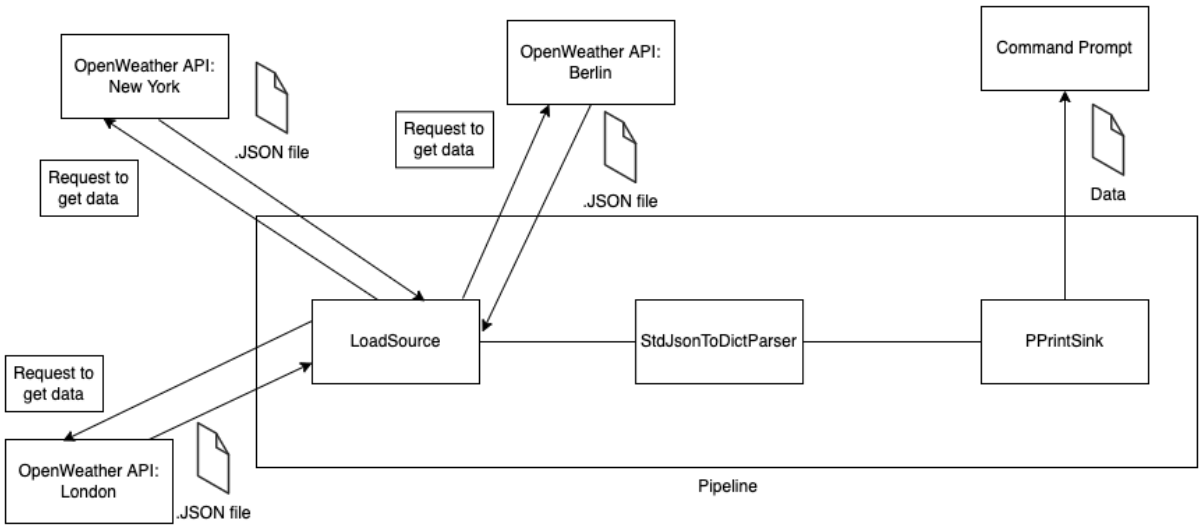
In this example we will learn how get data from one or multiple HTTP sources using an API request. In this case we cannot use basic HTTPClientSource, because it returns data only from one API query, so to get data from different queries we will have to define a new source for this use case.

The final pipeline will get data from multiple API requests in one time as a JSON, convert it to python dictionary, and output the data to Command Prompt.

In this example we will be using API from Open Weather to get current weather data (e.g, temperature, feels like temperature, biometric pressure etc).

In this example we will use `.conf` file to store configuration for our pump. More about how to write configuration is here *Configuration Quickstart*.

A diagram of the finished pipeline



Pipeline

In the code below you can see the structure of `SamplePipeline` which we need for this use case. The important part is the `self.build()` method where its parameters are the single components of the pipeline. Do not forget that every pipeline requires both source and sink to function correctly.

Source is a component that supply the pipeline with data. In our example we will use a specific type of source. Because we need to Pump data from API, we need to send request to the API to receive our data. This means that our source has to be "trigger" the request and send it to the API. For this reason we will be using a so-called trigger source. More about *Trigger Source*.

Because we are using *Trigger Source*. We need to specify which trigger we will be using. There are more types of triggers, but in our example we will be using PeriodicTrigger, which triggers in time intervals specified in the parameter. `bspump.trigger.PeriodicTrigger(app, <<Time parameter in seconds>>))`

Each pipeline requires a sink. In our example we want to see the result of the data, so we will be using PPrintSink which simply prints the data to the Command Prompt.

You can try to copy-paste this chunk of code and try it yourself. You must have BSPump module installed. Follow our guide *Installing BSPump module*.

Simply rewrite <<LOCATION>> to city you want to obtain data from and put your API key which you will get after you register on https://openweathermap.org/ to <<YOUR PRIVATE API KEY>> section. You can find more about how to modify your URL here **`https://openweathermap.org/current`_**

```
#!/usr/bin/env python3

import bspump
import bspump.common
import bspump.http
import bspump.trigger

class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            bspump.http.HTTPClientSource(app, self, config={
                'url': 'https://api.openweathermap.org/data/2.5/weather?q=<<LOCATION>>&
↪units=metric&appid=<<YOUR PRIVATE API KEY>>'
            }).on(bspump.trigger.PeriodicTrigger(app, 5)),
            bspump.common.PPrintSink(app, self),
        )

if __name__ == '__main__':
    app = bspump.BSPumpApplication()
    svc = app.get_service("bspump.PumpService")
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)
    app.run()
```

You should get output like this:

```
~python3 example.py
BitSwan BSPump version 21.11-17-g6b346fd
```

(continues on next page)

(continued from previous page)

```
27-Jan-2022 18:43:00.177421 NOTICE asab.application is ready.
1 pipeline(s) ready.
(b'{"coord":{"lon":-0.1257,"lat":51.5085},"weather":[{"id":802,"main":"Clouds",'
b'"description":"scattered clouds","icon":"03n"}],"base":"stations","main":{"t'
b'emp":8.91,"feels_like":6.86,"temp_min":6.8,"temp_max":10.14,"pressure":1030,'
b'"humidity":71},"visibility":10000,"wind":{"speed":3.6,"deg":290},"clouds":{"'
b'all":35},"dt":1643304840,"sys":{"type":2,"id":2019646,"country":"GB","sunris'
b'e":1643269577,"sunset":1643301595},"timezone":0,"id":2643743,"name":"London"'
b',"cod":200}')
```

Multiple locations source

In the code above, the pump simply returns data from one location. But in our use case we need to get data from multiple locations, which means we need to get data from multiple API's URL. Next, we define our specific trigger source.

We use ClientSession from aiohttp library to create session where get data from GET method as response for every city in our list. Then we store the data from response to event variable and process the event to pipeline. More about aiohttp session can be found here

```
class LoadSource(bspump.TriggerSource):

    def __init__(self, app, pipeline, choice=None, id=None, config=None):
        super().__init__(app, pipeline, id=id, config=config)
        self.cities = ['London','New York','Berlin'] #List of cities

    async def cycle(self):
        async with aiohttp.ClientSession() as session:
            #goes through the list of cities and requests from API for each city
            for city in self.cities:
                async with session.get(url=self.Config['url'].format(city=city, api_
↪key=self.Config['api_key'])) as response:
                    event = await response.content.read()
                    await self.process(event)
```

You can see that in this example we are using `self.Config` method to get the API key and the url from the configuration file. It is good to have the API key and the url in configuration file, because changes can be made simply in the configuration file.

For example, create a `weather-pump.conf` file, and into that file you can copy/paste code below

```
[pipeline:SamplePipeline:LoadSource]
url = https://api.openweathermap.org/data/2.5/weather?q={city}&units=metric&appid={api_
↪key}
api_key = <<YOUR PRIVATE API KEY>>
```

When you run your pump with configuration file you have to run it with `-c` switch. So after you finish your pump and you need to test it, type `python3 your-pump-name.py -c weather-pump.conf` to the terminal.

You can change the list of cities to any locations you wish. The important part of this source is `async def cycle(self)` method where we request the API's url for every location from our list and process them in the pipeline.

Just be sure that you import `aiohttp` package and change `HTTPClientSource` with our new specified `LoadSource`.

You can copy-paste the final code here:

```
#!/usr/bin/env python3

import bspump
import bspump.common
import bspump.http
import bspump.trigger
import aiohttp


class LoadSource(bspump.TriggerSource):

    def __init__(self, app, pipeline, choice=None, id=None, config=None):
        super().__init__(app, pipeline, id=id, config=config)
        self.cities = ['London','New York','Berlin'] #List of cities

    async def cycle(self):
        async with aiohttp.ClientSession() as session:
            #goes through the list of cities and requests from API for each city
            for city in self.cities:
                async with session.get(url=self.Config['url'].format(city=city, api_
↪key=self.Config['api_key'])) as response:
                    event = await response.content.read()
                    await self.process(event)


class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            LoadSource(app, self).on(
                bspump.trigger.PeriodicTrigger(app, 5)
            ),
            bspump.common.PPrintSink(app, self),
        )
if __name__ == '__main__':
        app = bspump.BSPumpApplication()
        svc = app.get_service("bspump.PumpService")
        pl = SamplePipeline(app, 'SamplePipeline')
        svc.add_pipeline(pl)
        app.run()
```

After you execute this code you should get this output in terminal:

```
~ python3 example.py -c example.conf
 BitSwan BSPump version 21.11-17-g6b346fd
 27-Jan-2022 18:56:14.058308 NOTICE asab.application is ready.
 1 pipeline(s) ready.
 (b'{"coord":{"lon":-0.1257,"lat":51.5085},"weather":[{"id":802,"main":"Clouds",'
 b'"description":"scattered clouds","icon":"03n"}],"base":"stations","main":{"t'
 b'emp":8.79,"feels_like":6.72,"temp_min":6.8,"temp_max":10.14,"pressure":1030,'
 b'"humidity":70},"visibility":10000,"wind":{"speed":3.6,"deg":290},"clouds":{"'
```

(continues on next page)

(continued from previous page)

```
b'all":35},"dt":1643305383,"sys":{"type":2,"id":2019646,"country":"GB","sunris'
b'e":1643269577,"sunset":1643301595},"timezone":0,"id":2643743,"name":"London"'
b',"cod":200}')
(b'{"coord":{"lon":-74.006,"lat":40.7143},"weather":[{"id":801,"main":"Clouds",'
b'"description":"few clouds","icon":"02d"}],"base":"stations","main":{"temp":-'
b'1.13,"feels_like":-1.13,"temp_min":-3.36,"temp_max":0.9,"pressure":1030,"hum'
b'idity":51},"visibility":10000,"wind":{"speed":0.45,"deg":34,"gust":1.34},"cl'
b'ouds":{"all":19},"dt":1643305980,"sys":{"type":2,"id":2039034,"country":"US"'
b',"sunrise":1643285428,"sunset":1643321212},"timezone":-18000,"id":5128581,"n'
b'ame":"New York","cod":200}')
(b'{"coord":{"lon":13.4105,"lat":52.5244},"weather":[{"id":803,"main":"Clouds",'
b'"description":"broken clouds","icon":"04n"}],"base":"stations","main":{"temp'
b'":6.01,"feels_like":1.09,"temp_min":5.01,"temp_max":6.85,"pressure":1003,"hu'
b'midity":91},"visibility":10000,"wind":{"speed":9.39,"deg":251,"gust":15.2},"'
b'clouds":{"all":75},"dt":1643305512,"sys":{"type":2,"id":2011538,"country":"D'
b'E","sunrise":1643266558,"sunset":1643298116},"timezone":3600,"id":2950159,"n'
b'ame":"Berlin","cod":200}')
```

Connect to ES

You can change and modify the pipeline in any manner you want. For example, instead of using PPrintSink you can use our Elasticsearch Sink which loads the data to Elasticsearch. If you want to read more about *How to connect to Elastic Search*.

2.2.4 Configuration Quickstart

In this tutorial you will learn about configuration in BSPump and how to use it.

What is configuration?

Every BitSwan object inside BSPump application can be configured using user-defined configuration options. It's good practice to write configuration in `.conf` files, because making changes will be much easier.

Every object has default configuration values set in `ConfigDefaults`. If you set `ConfigDefaults` in your specific class you override same values in `ConfigDefaults`, which are inherited from the parent class.

Configuration `.conf` files in are built-in in ASAB, the platform on which BSPump is built. You can find more about it in ASAB documentation

There are 3 methods to configure object

1. By defining `ConfigDefaults` dictionary inside specific class

```
class MySQLSource(TriggerSource):

    ConfigDefaults = {
            'query': 'SELECT id, name, surname FROM people;'
    }
```

2.Using `config` parameter in the object's constructor

```
bspump.mysql.MySQLSource(app, self, "MySQLConnection1", config={'query': 'SELECT id,␣
↪name, surname FROM people;'})
```

3. By creating `.conf` file

```
[pipeline:PipelineID]
query = SELECT id, name, surname FROM people;
```

Example

This example shows how to create a configuration file to get data from API via basic HTTPClientSource.

In first step we create .conf file where we store API key

```
[pipeline:SamplePipeline]
url = https://api.openweathermap.org/data/2.5/weather?q=London&units=metric&appid={api_
↪key}
api_key = <YOUR PRIVATE API KEY>
```

`[pipeline:SamplePipeline]` in this line we specify which class the configuration applies to. Values below this line override the same values in `ConfigDefaults` of specified classes.

Configuration in .conf file is accessible via self.Config method (in this case we use `self.Config['api_key']` to get API key from our `.conf` file)

In next step we have a sample pipeline that gets data through https://openweathermap.org/ API using API's URL and API key from .conf file. See more in coindesk.

```
#!/usr/bin/env python3

import bspump
import bspump.common
import bspump.http
import bspump.trigger


class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            bspump.http.HTTPClientSource(app, self,
            config={'url': self.Config['url'].format(api_key = self.Config['api_key'])}).
↪on(bspump.trigger.PeriodicTrigger(app, 2)),
            bspump.common.StdJsonToDictParser(app, self),
            bspump.common.PPrintSink(app, self)
        )


if __name__ == '__main__':
    app = bspump.BSPumpApplication()
```

(continues on next page)

(continued from previous page)

```
    svc = app.get_service("bspump.PumpService")
    # Construct and register Pipeline
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)

    app.run()
```

Running your pump with configuration files

When you want to run your pump with configuration file there are two ways to do that.

In terminal

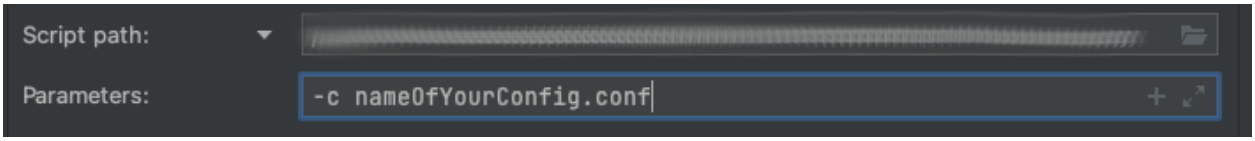
To run your pump with a configuration file, use `-c` switch in the terminal, after that switch there has to be `file_path/file_name.conf`.

For example when you have configuration file in same folder

```
~python3 mypumptest.py -c mypumpconfiguration.conf
```

In your IDE

To run your pump in IDE you have to set the run parameters. For example in PyCharm you have to go to Run -> Edit Configurations... and then change the run parameters to `-c file_path/nameOfYourConfig.conf`



2.2.5 How to connect to Elastic Search

BSPump supports the connection to Elastic Search platform. It is possible to connect to ES just in few lines of code.

Elastic Search Source

You can use Elastic Search Source to take data from Elastic Search index for further analysis over them (e.g. in your pump).

Prerequisites

You can access ElasticSearch only if you have ElasticSearch already installed on your server or you can try to install it locally with this tutorial *Install ElasticSearch and Kibana via Docker*.

The process of taking data from Elastic Search index is simple, you will need few things.

What you will need:

1. URL address of your Elastic Search
2. Index with data
3. Configuration file
4. Register the service of ESConnection

Configuration File

You will need to create `.conf` file with this configuration

```
# ElasticSearch Source
[pipeline:<<Name of your pipeline class>>:ElasticSearchSource]
index=<<Name of your index>>

# Elasticsearch connection
[connection:ESConnection]
url=<<Your ElasticSearch URL address>>
```

The configuration file can contain additional information depending on your implementation. It is essential to include: - *index* : name of the index that will be used to get data from - *url* : URL of your connection with ES

For more information visit our quickstart to using configs: *Configuration Quickstart*.

Code example

To create a connection with Elastic Search you will need to do two things:

1. Add ElasticSearchSource component to *self.build* method of the pipeline class
2. Add trigger which take data from index every defined time
3. create a service of your ES Connection.

You can implement your own ElasticSearch connection but the default connection will look like this:

```
import bspump.elasticsearch

class SamplePipeline(bspump.Pipeline):

def __init__(self, app, pipeline_id):
    super().__init__(app, pipeline_id)
    self.build(
        # Adding ES Source component with trigger set up to trigger data every 5 seconds
        bspump.elasticsearch.ElasticSearchSource(app, self, "ESConnection").on(bspump.
↪trigger.PeriodicTrigger(app, 5)),
        # Rest of the pipeline with source and processors
```

(continues on next page)

(continued from previous page)

```
    )

if __name__ == '__main__':
    app = bspump.BSPumpApplication()
    svc = app.get_service("bspump.PumpService")

    # Part where you add the ESConnection service
    es_connection = bspump.elasticsearch.ElasticSearchConnection(app, "ESConnection")
    svc.add_connection(es_connection)

    # Construct and register Pipeline
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)

    app.run()
```

It is important to include *"ESConnection"* as a parameter in ElasticSearch connection and source methods.

Elastic Search Sink

You can use Elastic Search sink to store data for further analysis or visualizations using Kibana.

Prerequisites

The process to create ES sink is simple and intuitive but you will need few things to start with.

What you will need:

1. URL address for connection with Elastic Search
2. Configuration file
3. Register the service of ESConnection

Configuration File

you will need to create `.conf` file using following syntax

```
# Elasticsearch connection
[connection:ESConnection]
url=<<YOUR CONNECTION URL>>

# Elasticsearch sink
[pipeline:<<Name of your pipeline class>>:ElasticSearchSink]
index=<<name of your index>>
doctype=_doc
```

The configuration file can contain additional information depending on your implementation. It is essential to include:

- *index* : name of the index that will be used to store your data in ES
- *url* : URL of your connection with ES
- *doctype* : type of the document, default is *_doc*

For more information visit our quickstart to using configs: *Configuration Quickstart*.

Code example

To create a connection with Elastic Search you will need to do two things:

1. Add ElasticSearchSink component to *self.build* method of the pipeline class
2. create a service of your ES Connection.

You can implement your own ElasticSearch connection but the default connection will look like this:

```
import bspump.elasticsearch

class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)
        self.build(
            #Rest of the pipeline with source and processors
            #Adding ES Sink component
            bspump.elasticsearch.ElasticSearchSink(app, self, "ESConnection"),
        )

if __name__ == '__main__':
    app = bspump.BSPumpApplication()
    svc = app.get_service("bspump.PumpService")

    #Part where you add the ESConnection service
    es_connection = bspump.elasticsearch.ElasticSearchConnection(app, "ESConnection")
    svc.add_connection(es_connection)

    svc.add_connection(
        bspump.kafka.KafkaConnection(app, "KafkaConnection")
    )

    app.run()
```

It is important to include *"ESConnection"* as a parameter in ElasticSearch connection and sink methods.

2.2.6 Escape From Tarkov Craft Profit Counter

About

Pipeline in this example is inspired by game Escape from Tarkov. It is a realistic FPS game. Beside shooting enemies, Players can earn and sell items in a game market which is driven by players themselves. The price of each item is changing in real-time based on Demand-supply mechanics. Another important game mechanic is that players can create the items themself in their specific stations. Items created and required for each crafts can be bought on the market, so players can earn in-game money by producing the items. Because price of each item is not stable, some crafts are more profitable than others. My idea was to take data from an API source that gives information of all available crafts players can do together with price of each item. I will use this data to sort and analyze the data and output them in form that might help players know which crafts is more profitable and suitable.

In this example I will show you a process of creating pipeline with a bit more complicated use. You will learn about creating a source that enables us to use query in our API requests.

Source

First we have to create our source to pump the data to the pipeline. We will be using aiohttp library for our custom source. We will start by creating our source class. As you can see in the code below.

```
class IOHTTPSource(bspump.TriggerSource):
    def __init__(self, app, pipeline, choice=None, id=None, config=None):
        super().__init__(app, pipeline, id=id, config=config)

    async def cycle(self):
        async with aiohttp.ClientSession() as session:
            async with session.post('https://tarkov-tools.com/graphql', json={'query':␣
↪query}) as response:
                if response.status == 200:
                    event = await response.json()
                else:
                    raise Exception("Query failed to run by returning code of {}. {}".
↪format(response.status, query))
                await self.process(event)
```

As you can see in the cycle method. We are using asynchronous functions for the API requests. As you can see in the code I am creating Session which is used in aiohttp for more information check AIOHTTP Documentation. I am using post method with a query parameter as seen below.

```
    query = """
query {
  crafts {
    source
    duration
    rewardItems {
      quantity
      item {
        shortName
        lastLowPrice
      }
    }
    requiredItems {
      quantity
      item {
        shortName
        lastLowPrice
      }
    }
  }
}
"""
```

I created this query using playground interface made by the API authors. Here is the link if you would like to use this API.

Now you can try to copy-paste the code below and try it for yourself.

```
#!/usr/bin/env python3
import aiohttp
import bspump
import bspump.common
import bspump.http
import bspump.trigger
import pandas as pd
import bspump.file

query = """
query {
  crafts {
    source
    duration
    rewardItems {
      quantity
      item {
        shortName
        lastLowPrice
      }
    }
    requiredItems {
      quantity
      item {
        shortName
        lastLowPrice
      }
    }
  }
}
"""

class IOHTTPSource(bspump.TriggerSource):
    def __init__(self, app, pipeline, choice=None, id=None, config=None):
        super().__init__(app, pipeline, id=id, config=config)

    async def cycle(self):
        async with aiohttp.ClientSession() as session:
            async with session.post('https://tarkov-tools.com/graphql', json={'query':␣
↪query}) as response:
                if response.status == 200:
                    event = await response.json()
                else:
                    raise Exception("Query failed to run by returning code of {}. {}".
↪format(response.status, query))
                await self.process(event)


class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)
```

(continues on next page)

(continued from previous page)

```
        self.build(
            IOHTTPSource(app, self).on(bspump.trigger.PeriodicTrigger(app, 5)),
            bspump.common.PPrintProcessor(app,self),
            bspump.common.NullSink(app, self),
        )
```

If everything works correctly, you should be getting similar output.

```
'source': 'Workbench level 3'},
{'duration': 60000,
'requiredItems': [{'item': {'lastLowPrice': 39000,
                            'shortName': 'Eagle'},
                   'quantity': 2},
                  {'item': {'lastLowPrice': 15000,
                            'shortName': 'Kite'},
                   'quantity': 2}],
'rewardItems': [{'item': {'lastLowPrice': None,
                          'shortName': 'BP'},
                 'quantity': 120}],
'source': 'Workbench level 3'},
{'duration': 61270,
'requiredItems': [{'item': {'lastLowPrice': 15000,
                            'shortName': 'Kite'},
                   'quantity': 2},
                  {'item': {'lastLowPrice': 39000,
                            'shortName': 'Eagle'},
                   'quantity': 2},
                  {'item': {'lastLowPrice': 31111,
                            'shortName': 'Hawk'},
                   'quantity': 2}],
'rewardItems': [{'item': {'lastLowPrice': None,
                          'shortName': 'PPBS'},
                 'quantity': 150}],
                      .
                      .
                      .
```

There are probably hundreds of JSON lines in your console right now. It is not a nice way to output your data right? Let's implement our filter processor then.

Filter Processor

This filter processor is used for very specific use-case in this example. The goal as you can remember was to filter incoming data. The goal is to create a dataframe that contains data where each row has information about station in which the craft is created, duration of the craft ,price of items needed to perform the craft, name and price of item/s that we obtain by the craft, profit of the craft, and profit per hour. As you can see there is a lot of indexes we have to create.

```
class FilterByStation(bspump.Processor):
    def __init__(self, app, pipeline, id=None, config=None):
        super().__init__(app, pipeline, id=None, config=None)
```

(continues on next page)

(continued from previous page)

```
    def process(self, context, event):
        my_columns = ['station', 'name', 'output_price_item', 'duration', 'input_price_
↪item', 'profit', 'profit_per_hour']
        df = pd.DataFrame(columns=my_columns)
        for item in event["data"]["crafts"]:
            duration = round((item["duration"])/60/60, ndigits=3)
            reward = item["rewardItems"][0]
            name_output = reward["item"]["shortName"]
            quantity = reward["quantity"]
            output_item_price = reward["item"]["lastLowPrice"]
            if output_item_price is None:  # checks for NULL values
                output_item_price = 0
            output_price_item = quantity * int(output_item_price)
            station_name = item["source"]
            profit = 0
            profit_p_hour = 0
            input_price_item = 0
            for item2 in range(len(item["requiredItems"])):
                required_item = item["requiredItems"][item2]
                quantity_i = required_item["quantity"]
                input_item = required_item["item"]["lastLowPrice"]
                if input_item is None:
                    input_item = 0
                price_of_input_item = input_item * quantity_i
                input_price_item = input_price_item + price_of_input_item
                profit = output_price_item - input_price_item
                profit_p_hour = round(profit / duration, ndigits=3)
            df = df.append(
                pd.Series([station_name,
                           name_output,
                           output_price_item,
                           duration,
                           input_price_item,
                           profit,
                           profit_p_hour],
                          index=my_columns), ignore_index=True)
            event = df
        return event
```

You can copy-paste the code above and everything should work just fine. Don't forget to reference the processor in the self.build() method.

```
class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            IOHTTPSource(app, self).on(bspump.trigger.PeriodicTrigger(app, 5)),
            FilterByStation(app, self),
            bspump.common.PPrintProcessor(app, self),
            bspump.common.NullSink(app, self),
```

(continues on next page)

(continued from previous page)

```
        )
```

If you want more detail of what it does. It firstly goes through the whole json, then it gets data for each of the index if possible (otherwise zero is used instead of null), and appends the record as a row in our dataframe. I am using Pandas in this example. If you are not familiar with Pandas make sure you checked their Documentation

Now output in your console should like like this:

```
                        station         name output_price_item  duration input_price_
↪item   profit  profit_per_hour
0        Booze generator level 1    Moonshine             286999     3.056          ␣
↪236998    50001        16361.584
1    Intelligence Center level 2  Flash drive             180000    34.222          ␣
↪151498    28502          832.856
2    Intelligence Center level 2       Virtex              88000    37.611          ␣
↪210993  -122993        -3270.134
3    Intelligence Center level 2       SG-C10             130000    38.889          ␣
↪206978   -76978        -1979.429
4    Intelligence Center level 2        RFIDR             215000    53.333           ␣
↪40000   175000         3281.271
..                          ...          ...                ...       ...              ..
↪.      ...                 ...
128            Workbench level 3          PBP                  0    11.972          ␣
↪265888  -265888       -22209.155
129            Workbench level 3         M995                  0    15.994          ␣
↪211000  -211000       -13192.447
130            Workbench level 3          M61                  0    16.644          ␣
↪233331  -233331       -14018.926
131            Workbench level 3           BP                  0    16.667          ␣
↪108000  -108000        -6479.870
132            Workbench level 3         PPBS                  0    17.019          ␣
↪170222  -170222       -10001.880

[133 rows x 7 columns]
```

We can agree that this looks much more better than raw JSON, but this is not the end we still need to send the data somewhere for out bot

Dataframe to csv Processor

To make the data available for our Discord bot, we will save them to a directory as a csv file. This processor is really simple as we call only one function from the Pandas library.

You can copy paste the code of the processor

```
class DataFrameToCSV(bspump.Processor):
    def __init__(self, app, pipeline, id=None, config=None):
        super().__init__(app, pipeline, id=None, config=None)

    def process(self, context, event):
        event.to_csv('./Data/TarkovData.csv', index=False)
        return event
```

Once again dont forget to include the processor in our self.build() method.

```
class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            IOHTTPSource(app, self).on(bspump.trigger.PeriodicTrigger(app, 5)),
            FilterByStation(app, self),
            bspump.common.PPrintProcessor(app, self),
            DataFrameToCSV(app, self),
            bspump.common.NullSink(app, self),
        )
```

This wont change our output in console, but it should create a csv file in your current directory.

What next

Now we have a function pipeline. You can do anything with the output data. For example, I created a simple discord bot that sends a message with the updated data you can try to make your own discord bot using this tutorial: Getting Started with Discord Bots.

2.2.7 Fortnite Current Store Example

About

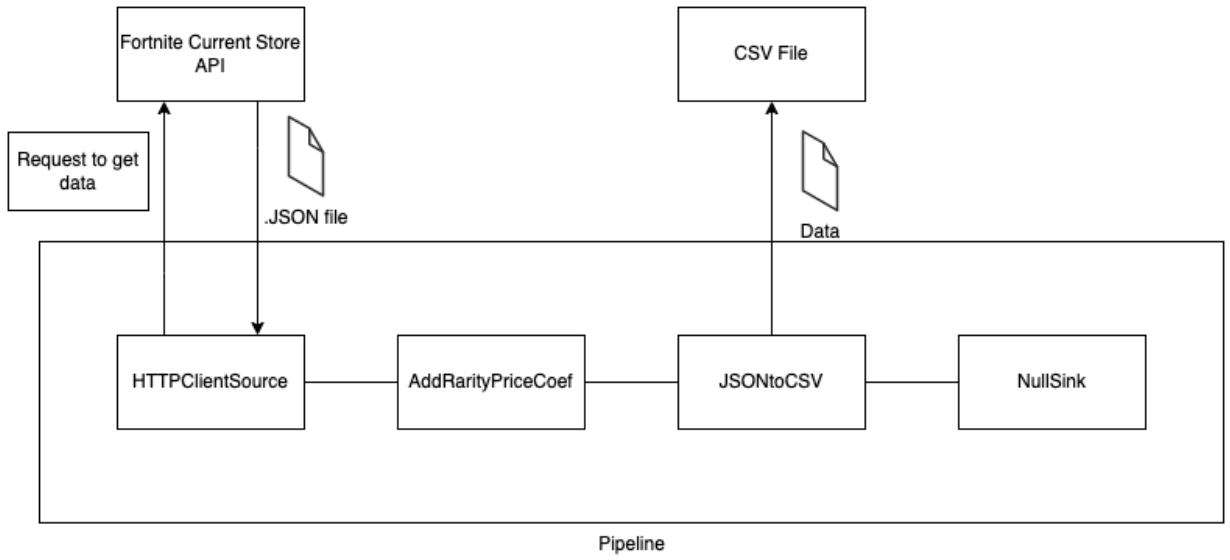
In this example we will get data from one HTTP course using an API request and use filtering processors on those datas and export the data to `.csv` file which can be used for example for Discord bot.

The final pipeline will get data form API request, filter some values from dataframe, does some calculation with values and then export it to CSV file.

We will be using API from Fortnite Tracker to get current Fortnite store items.

We will work with configuration files in this example. If you already doesn't know how to work with configuration files try this quickstart *Configuration Quickstart*.

A diagram of the finished pipeline



First sample pipeline

In the code below you can see the structure of `SamplePipeline` which we need for this use case. The important part is the `self.build()` method where its parameters are the single components of the pipeline. Do not forget that every pipeline requires both source and sink to function correctly.

Source is a component that supply the pipeline with data. In our example we will use a specific type of source. Because we need to Pump data from API, we need to send request to the API to receive our data. This means that our source has to be "trigger" the request and send it to the API. For this reason we will be using a so-called trigger source. More about *Trigger Source*.

Because we are using *Trigger Source*. We need to specify which trigger we will be using. There are more types of triggers, but in our example we will be using PeriodicTrigger, which triggers in time intervals specified in the parameter. `bspump.trigger.PeriodicTrigger(app, <<Time parameter in seconds>>))`

Each pipeline requires a sink. We will use PPrintSink for now to see incoming data. But in the next steps we will be using NullSink which I describe later.

First we need to create configuration file. Create `config.conf` file in your pump folder. To this configuration file copy-paste this chunk of code and rewrite `<YOUR PRIVATE API>` section with your API key which you will get by following steps here

```
[pipeline:SamplePipeline]
url = https://api.fortnitetracker.com/v1/store
api_key = <YOUR PRIVATE API KEY>
```

After you have your configuration file finished you can copy-paste code below and try it yourself. Be sure you have BSPump module installed. If not follow our guide *Installing BSPump module*

```
import bspump
import bspump.common
import bspump.http
import bspump.trigger

class SamplePipeline(bspump.Pipeline):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)
        self.build(
            bspump.http.HTTPClientSource(app, self,
            config={'url': self.Config['url']},
            headers={'TRN-Api-Key': self.Config['api_key']}).on(bspump.trigger.
↪PeriodicTrigger(app, 2)),
            bspump.common.PPrintSink(app, self),
        )


if __name__ == '__main__':
    app = bspump.BSPumpApplication()

    svc = app.get_service("bspump.PumpService")

    # Construct and register Pipeline
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)

    app.run()
```

You can run this code with ~ `python3 yourpumpname.py -c config.conf` command in terminal.

Well done! Now we are pumping data about items which are in Fortnite store right now.

You should get output like this:

```
~ python3 docs1.py -c config.conf
BitSwan BSPump version 21.11-17-g6b346fd
04-Feb-2022 18:00:30.503021 NOTICE asab.application is ready.
1 pipeline(s) ready.
(b'[\r\n  {\r\n    "imageUrl": "https://trackercdn.com/legacycdn/fortnite/8BD06'
 b'909_large.png",\r\n    "manifestId": 6909,\r\n    "name": "Marsh Walk",\r'
 b'\n    "rarity": "Sturdy",\r\n    "storeCategory": "BRSpecialFeatured",\r'
 b'\n    "vBucks": 500\r\n  },\r\n  {\r\n    "imageUrl": "https://trackercdn.c'
 b'om/legacycdn/fortnite/275915210_large.png",\r\n    "manifestId": 15210,\r\n '
 b'   "name": "Arcane Vi",\r\n    "rarity": "Epic",\r\n    "storeCategory": "BR'
 b'SpecialFeatured",\r\n    "vBucks": 0\r\n  },\r\n  {\r\n    "imageUrl": "http'
 b's://trackercdn.com/legacycdn/fortnite/2AC415212_large.png",\r\n    "manife'
 b'stId": 15212,\r\n    "name": "Piltover Warden Hammer",\r\n    "rarity": "Epi'
 b'c",\r\n    "storeCategory": "BRSpecialFeatured",\r\n    "vBucks": 800\r\n '
 b' },\r\n  {\r\n    "imageUrl": "https://trackercdn.com/legacycdn/fortnite/6C4'
 b'015364_large.png",\r\n    "manifestId": 15364,\r\n    "name": "Marsha",\r'
 b'\n    "rarity": "Epic",\r\n    "storeCategory": "BRSpecialFeatured",\r\n '
 b'   "vBucks": 1500\r\n  },\r\n  {\r\n    "imageUrl": "https://trackercdn.co'
 b'm/legacycdn/fortnite/46F66923_large.png",\r\n    "manifestId": 6923,\r\n    '
 b'"name": "Marshmello",\r\n    "rarity": "Quality",\r\n    "storeCategory": "B'
 b'RSpecialFeatured",\r\n    "vBucks": 1500\r\n  },\r\n  {\r\n    "imageUrl": "'
 b'https://trackercdn.com/legacycdn/fortnite/B84F13565_large.png",\r\n    "ma'
 b'nifestId": 13565,\r\n    "name": "Arcane Jinx",\r\n    "rarity": "Epic","
 b'\r\n    "storeCategory": "BRSpecialFeatured",\r\n    "vBucks": 0\r\n  },\r\n'
 b'  {\r\n    "imageUrl": "https://trackercdn.com/legacycdn/fortnite/61841528'
```

(continues on next page)

(continued from previous page)

```
b'7_large.png",\r\n    "manifestId": 15287,\r\n    "name": "Goblin Glider"'
b',\r\n    "rarity": "Epic",\r\n    "storeCategory": "BRSpecialFeatured",\r'
b'\n    "vBucks": 800\r\n  },\r\n ...
```

Export to CSV

Awesome! Now we are pumping data but we want to store them somewhere. In the end we want to create Discord Bot which will show us current Fortnite Store when we write command to discord chat. Discord bot can work easily with CSV file so we need to export our data do *.csv* file.

We have to import *pandas* library to our pump which can export JSON file to CSV file and then we define our exporting processor.

The processor convert JSON file to dataframe with pandas library and then export it as CSV file and create specified file in same folder like our pump (you can define path you want).

This will be our processor:

```
class JSONtoCSV(bspump.Processor):

    def process(self, context, event):
        df = pd.read_json(event)
        event = df.to_csv('data.csv', index=False)
        return event
```

Now we add this processor to our pump, we have to change PPrintSink to NullSink because we don't want to store or print data anywhere, we will have it in our CSV file.

You can copy-paste code below and look into your pump folder if there is a CSV file with our data.

```
import bspump
import bspump.common
import bspump.http
import bspump.trigger
import pandas as pd

class JSONtoCSV(bspump.Processor):

        def process(self, context, event):
            df = pd.read_json(event)
            event = df.to_csv('data.csv', index=False)
            return event

class SamplePipeline(bspump.Pipeline):
    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            bspump.http.HTTPClientSource(app, self,
            config={'url': self.Config['url']},
            headers={'TRN-Api-Key': self.Config['api_key']}).on(bspump.trigger.
↪PeriodicTrigger(app, 2)),
            JSONtoCSV(app, self),
```

(continues on next page)

(continued from previous page)

```
            bspump.common.NullSink(app, self),
        )


if __name__ == '__main__':
    app = bspump.BSPumpApplication()

    svc = app.get_service("bspump.PumpService")

    # Construct and register Pipeline
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)

    app.run()
```

The CSV file should looks this way:

data

imageUrl	manifestId	name	rarity	storeCategory	vBucks
https://trackercdn.com/legacycdn/fortnite/8BD06909_large.png	6909	Marsh Walk	Sturdy	BRSpecialFeatured	500
https://trackercdn.com/legacycdn/fortnite/275915210_large.png	15210	Arcane Vi	Epic	BRSpecialFeatured	0
https://trackercdn.com/legacycdn/fortnite/2AC415212_large.png	15212	Piltover Warden Hammer	Epic	BRSpecialFeatured	800
https://trackercdn.com/legacycdn/fortnite/6C4015364_large.png	15364	Marsha	Epic	BRSpecialFeatured	1500
https://trackercdn.com/legacycdn/fortnite/46F66923_large.png	6923	Marshmello	Quality	BRSpecialFeatured	1500
https://trackercdn.com/legacycdn/fortnite/B84F13565_large.png	13565	Arcane Jinx	Epic	BRSpecialFeatured	0
https://trackercdn.com/legacycdn/fortnite/618415287_large.png	15287	Goblin Glider	Epic	BRSpecialFeatured	800
https://trackercdn.com/legacycdn/fortnite/A3BF15367_large.png	15367	MARSHINOBI	Epic	BRSpecialFeatured	1600
https://trackercdn.com/legacycdn/fortnite/6B6115288_large.png	15288	Arm the Pumpkin!	Epic	BRSpecialFeatured	200
https://trackercdn.com/legacycdn/fortnite/475613566_large.png	13566	Pow Pow Crusher	Epic	BRSpecialFeatured	800
https://trackercdn.com/legacycdn/fortnite/DE1815366_large.png	15366	Maximum Bounce	Rare	BRSpecialFeatured	500
https://trackercdn.com/legacycdn/fortnite/4F7E13567_large.png	13567	Playground (Instrumental)	Rare	BRSpecialFeatured	200
https://trackercdn.com/legacycdn/fortnite/AAE715285_large.png	15285	Pumpkin P'axe	Epic	BRSpecialFeatured	800
https://trackercdn.com/legacycdn/fortnite/5E246922_large.png	6922	Mello Rider	Handmade	BRSpecialFeatured	500
https://trackercdn.com/legacycdn/fortnite/2FC715211_large.png	15211	Punching Practice	Epic	BRSpecialFeatured	200
https://trackercdn.com/legacycdn/fortnite/3FF311949_large.png	11949	Mello Mallets	Rare	BRSpecialFeatured	800
https://trackercdn.com/legacycdn/fortnite/ABE515286_large.png	15286	Green Goblin	Epic	BRSpecialFeatured	0
https://trackercdn.com/legacycdn/fortnite/742015369_large.png	15369	Mello Glo	Epic	BRSpecialFeatured	800
https://trackercdn.com/legacycdn/fortnite/7AD414623_large.png	14623	Azuki	Rare	BRWeeklyStorefront	1400
https://trackercdn.com/legacycdn/fortnite/9D7912776_large.png	12776	Hi-Octane	rare	BRWeeklyStorefront	800
https://trackercdn.com/legacycdn/fortnite/70C012877_large.png	12877	Hedron	epic	BRWeeklyStorefront	1500
https://trackercdn.com/legacycdn/fortnite/7EED12221_large.png	12221	Black Ooze	Rare	BRWeeklyStorefront	500
https://trackercdn.com/legacycdn/fortnite/1F2E12775_large.png	12775	Pitstop	rare	BRWeeklyStorefront	1200
https://trackercdn.com/legacycdn/fortnite/B14212875_large.png	12875	Pick Axis	rare	BRWeeklyStorefront	800
https://trackercdn.com/legacycdn/fortnite/613F12774_large.png	12774	Storm Racer	rare	BRWeeklyStorefront	1200
https://trackercdn.com/legacycdn/fortnite/768012220_large.png	12220	Chaos Scythe	Rare	BRWeeklyStorefront	800
https://trackercdn.com/legacycdn/fortnite/45E212219_large.png	12219	Chaos Agent	Epic	BRWeeklyStorefront	1500
https://trackercdn.com/legacycdn/fortnite/BAD912874_large.png	12874	Iso	epic	BRWeeklyStorefront	1500
https://trackercdn.com/legacycdn/fortnite/AC4012876_large.png	12876	Multipoint Edge	rare	BRWeeklyStorefront	800
https://trackercdn.com/legacycdn/fortnite/2BED12384_large.png	12384	Poki	Rare	BRDailyStorefront	500
https://trackercdn.com/legacycdn/fortnite/6A8F13318_large.png	13318	Bear Hug	uncommon	BRDailyStorefront	200
https://trackercdn.com/legacycdn/fortnite/DBD85393_large.png	5393	Eagle	Sturdy	BRDailyStorefront	500
https://trackercdn.com/legacycdn/fortnite/676E12368_large.png	12368	Monks	Rare	BRDailyStorefront	1200
https://trackercdn.com/legacycdn/fortnite/379B12636_small.png	12636	The Renegade	rare	BRDailyStorefront	500
https://trackercdn.com/legacycdn/fortnite/AF2C6888_large.png	6888	Volley Girl	Sturdy	BRDailyStorefront	1200

Processor with pandas script

You can see that in our data set there aren't so many interesting datas. So we want to add column with coefficient of price over rarity which will be useful in our Discord bot, because player could know which items is the most advantageous for purchase.

We create basic pandas script to go through rows and calculate the coefficient from rarity and vBucks column values and then add to list which will create new column called *Coef* at the end. More about pandas here

You have to convert the dataframe back to JSON file, because pipeline can't work with dataframes.

The processor:

```
class AddRarityPriceCoef(bspump.Processor):

    def process(self, context, event):
        df = pd.read_json(event)
        coefs = []
        for row in df.itertuples():
            if row.vBucks == 0:
                price = 1
            else:
                price = row.vBucks
            if row.rarity.lower() == 'handmade':
                coefs.append((1/price)*100)
            elif row.rarity.lower() == 'uncommon':
                coefs.append((2/price)*100)
            elif row.rarity.lower() == 'rare':
                coefs.append((3/price)*100)
            elif row.rarity.lower() == 'epic':
                coefs.append((4/price)*100)
            elif row.rarity.lower() == 'legendary':
                coefs.append((5/price)*100)
            elif row.rarity.lower() == 'mythic':
                coefs.append((6/price)*100)
            elif row.rarity.lower() == 'exotic':
                coefs.append((7/price)*100)
            else:
                coefs.append(1)
        df['Coef'] = coefs
        event = df.to_json()
        return event
```

Now we add the processor to our pump and after you copy-paste the code and run the pump you can see that the new column was added with our calculated values.

```
#!/usr/bin/env python3

import bspump
import bspump.common
import bspump.http
import bspump.trigger
import pandas as pd


class JSONtoCSV(bspump.Processor):

    def process(self, context, event):
        df = pd.read_json(event)
        print(df)
        event = df.to_csv('data.csv', index=False)
        return event


class AddRarityPriceCoef(bspump.Processor):
```

(continues on next page)

(continued from previous page)

```
    def process(self, context, event):
        df = pd.read_json(event)
        coefs = []
        for row in df.itertuples():
            if row.vBucks == 0:
                price = 1
            else:
                price = row.vBucks
            if row.rarity.lower() == 'handmade':
                coefs.append((1/price)*100)
            elif row.rarity.lower() == 'uncommon':
                coefs.append((2/price)*100)
            elif row.rarity.lower() == 'rare':
                coefs.append((3/price)*100)
            elif row.rarity.lower() == 'epic':
                coefs.append((4/price)*100)
            elif row.rarity.lower() == 'legendary':
                coefs.append((5/price)*100)
            elif row.rarity.lower() == 'mythic':
                coefs.append((6/price)*100)
            elif row.rarity.lower() == 'exotic':
                coefs.append((7/price)*100)
            else:
                coefs.append(1)
        df['Coef'] = coefs
        event = df.to_json()
        return event


class SamplePipeline(bspump.Pipeline):
    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)
        self.build(
            bspump.http.HTTPClientSource(app, self,
            config={'url': self.Config['url']},
            headers={'TRN-Api-Key': self.Config['api_key']}).on(bspump.trigger.
↪PeriodicTrigger(app, 2)),
            # Add price over rarity coefficient to dataframe
            AddRarityPriceCoef(app, self),
            # Converts incoming json event to CSV data
            JSONtoCSV(app, self),
            # We can also push datas to ES or Kafka
            bspump.common.NullSink(app, self),
        )


if __name__ == '__main__':
    app = bspump.BSPumpApplication()

    svc = app.get_service("bspump.PumpService")

    # Construct and register Pipeline
```

(continues on next page)

(continued from previous page)

```
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)

    app.run()
```

Data in CSV file:

data

imageUrl	manifestId	name	rarity	storeCategory	vBucks	Coef
https://trackercdn.com/legacycdn/fortnite/8BD06909_large.png	6909	Marsh Walk	Sturdy	BRSpecialFeatured	500	0.6000000000000001
https://trackercdn.com/legacycdn/fortnite/275915210_large.png	15210	Arcane Vi	Epic	BRSpecialFeatured	0	400.0
https://trackercdn.com/legacycdn/fortnite/2AC415212_large.png	15212	Piltover Warden Hammer	Epic	BRSpecialFeatured	800	0.5
https://trackercdn.com/legacycdn/fortnite/6C4015364_large.png	15364	Marsha	Epic	BRSpecialFeatured	1500	0.2666666667
https://trackercdn.com/legacycdn/fortnite/46F66923_large.png	6923	Marshmello	Quality	BRSpecialFeatured	1500	0.2666666667
https://trackercdn.com/legacycdn/fortnite/B84F13565_large.png	13565	Arcane Jinx	Epic	BRSpecialFeatured	0	400.0
https://trackercdn.com/legacycdn/fortnite/618415287_large.png	15287	Goblin Glider	Epic	BRSpecialFeatured	800	0.5
https://trackercdn.com/legacycdn/fortnite/A3BF15367_large.png	15367	MARSHINOBI	Epic	BRSpecialFeatured	1600	0.25
https://trackercdn.com/legacycdn/fortnite/6B6115288_large.png	15288	Arm the Pumpkin!	Epic	BRSpecialFeatured	200	2.0
https://trackercdn.com/legacycdn/fortnite/475613566_large.png	13566	Pow Pow Crusher	Epic	BRSpecialFeatured	800	0.5
https://trackercdn.com/legacycdn/fortnite/DE1815366_large.png	15366	Maximum Bounce	Rare	BRSpecialFeatured	500	0.6000000000000001
https://trackercdn.com/legacycdn/fortnite/4F7E13567_large.png	13567	Playground (Instrumental)	Rare	BRSpecialFeatured	200	1.5
https://trackercdn.com/legacycdn/fortnite/AAE715285_large.png	15285	Pumpkin P'axe	Epic	BRSpecialFeatured	800	0.5
https://trackercdn.com/legacycdn/fortnite/5E246922_large.png	6922	Mello Rider	Handmade	BRSpecialFeatured	500	0.2
https://trackercdn.com/legacycdn/fortnite/2FC715211_large.png	15211	Punching Practice	Epic	BRSpecialFeatured	200	2.0
https://trackercdn.com/legacycdn/fortnite/3FF311949_large.png	11949	Mello Mallets	Rare	BRSpecialFeatured	800	0.375
https://trackercdn.com/legacycdn/fortnite/ABE515286_large.png	15286	Green Goblin	Epic	BRSpecialFeatured	0	400.0
https://trackercdn.com/legacycdn/fortnite/742015369_large.png	15369	Mello Glo	Epic	BRSpecialFeatured	800	0.5
https://trackercdn.com/legacycdn/fortnite/7AD414623_large.png	14623	Azuki	Rare	BRWeeklyStorefront	1400	0.2142857143
https://trackercdn.com/legacycdn/fortnite/9D7912776_large.png	12776	Hi-Octane	rare	BRWeeklyStorefront	800	0.375
https://trackercdn.com/legacycdn/fortnite/70C012877_large.png	12877	Hedron	epic	BRWeeklyStorefront	1500	0.2666666667
https://trackercdn.com/legacycdn/fortnite/7EED12221_large.png	12221	Black Ooze	Rare	BRWeeklyStorefront	500	0.6000000000000001
https://trackercdn.com/legacycdn/fortnite/1F2E12775_large.png	12775	Pitstop	rare	BRWeeklyStorefront	1200	0.25
https://trackercdn.com/legacycdn/fortnite/B14212875_large.png	12875	Pick Axis	rare	BRWeeklyStorefront	800	0.375
https://trackercdn.com/legacycdn/fortnite/613F12774_large.png	12774	Storm Racer	rare	BRWeeklyStorefront	1200	0.25
https://trackercdn.com/legacycdn/fortnite/768012220_large.png	12220	Chaos Scythe	Rare	BRWeeklyStorefront	800	0.375
https://trackercdn.com/legacycdn/fortnite/45E212219_large.png	12219	Chaos Agent	Epic	BRWeeklyStorefront	1500	0.2666666667
https://trackercdn.com/legacycdn/fortnite/BAD912874_large.png	12874	Iso	epic	BRWeeklyStorefront	1500	0.2666666667
https://trackercdn.com/legacycdn/fortnite/AC4012876_large.png	12876	Multipoint Edge	rare	BRWeeklyStorefront	800	0.375
https://trackercdn.com/legacycdn/fortnite/2BED12384_large.png	12384	Poki	Rare	BRDailyStorefront	500	0.6000000000000001
https://trackercdn.com/legacycdn/fortnite/6A8F13318_large.png	13318	Bear Hug	uncommon	BRDailyStorefront	200	1.0
https://trackercdn.com/legacycdn/fortnite/DBD85393_large.png	5393	Eagle	Sturdy	BRDailyStorefront	500	0.6000000000000001
https://trackercdn.com/legacycdn/fortnite/676E12368_large.png	12368	Monks	Rare	BRDailyStorefront	1200	0.25
https://trackercdn.com/legacycdn/fortnite/379B12636_small.png	12636	The Renegade	rare	BRDailyStorefront	500	0.6000000000000001
https://trackercdn.com/legacycdn/fortnite/AF2C6888_large.png	6888	Volley Girl	Sturdy	BRDailyStorefront	1200	0.25

Conclusion

So, in this example we learnt how to get data from basic API request and export it to CSV file. Then we create script with pandas library to make price over rarity coefficient and add it as a new column to our dataset. You can also add some other processors which can filter data or make some calculation over the datas.

What next?

Now I will show you how can you use the pump to create your Discord bot for yourself or your friends.

You can find how to create Discord bot here

The following discord bot can looks like this:

2.2.8 Install ElasticSearch and Kibana via Docker

About

This example is focused on how to install ElasticSearch and Kibana on your localhost and use the ES via Kibana GUI. We will be using Docker and Docker compose to install ElasticSearch environment. Be sure you have set up Docker and Docker compose, if not follow this guide to install Docker and Docker compose.

In the end we use Docker image of our Weather Pump, which can be found here *Weather API Example*, to pump data to index in our local ElasticSearch.

Docker is a platform which provides the ability to package and run an application in a loosely isolated environment called a container. More about Docker you can also read our quickstart how to use Docker with BSPump module here: *Docker File Quickstart*

Docker compose is a tool for defining and running multi-container Docker applications. More about Docker compose.

Docker compose with ES and Kibana

Now we create Docker compose file to run ElasticSearch and Kibana on our localhost. Create `docker-compose.yml` file in your specified folder. In docker compose you have to define your services which you want to use. In our case we define `elasticsearch` and `kibana`. We choose which image of ES and Kibana we want to use. The image will automatically download from official Docker hub of Elastic. Then we set a names of container and set a condition when the container restart after unexpected exit. In next step we set the environment of container. In this case we don't want to have security, we will have just one ElasticSearch single-node and we set up a connection between ES and Kibana in `ELASTICSEARCH_HOSTS`. Volumes is where the data will be stored in container file system. And in the end we specified on which localhost port container will be running. You can also set that one service will be wait for another in `depends_on`.

Just copy-paste this chunk of code into your `docker-compose.yml` file:

```
version: '3.9'
services:
  # Elastic Search single node cluster
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.0.0
    container_name: elasticsearch
    restart: always
```

(continues on next page)

(continued from previous page)

```
    environment:
      - xpack.security.enabled=false
      - discovery.type=single-node
    volumes:
      - elasticsearch-data-volume:/usr/share/elasticsearch/data/
    ports:
      - 9200:9200
      - 9300:9300
  # Kibana UI for Elastic Search
  kibana:
    image: docker.elastic.co/kibana/kibana:8.0.0
    container_name: kibana
    restart: always
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    ports:
      - 5601:5601
    depends_on:
      - elasticsearch

volumes:
  elasticsearch-data-volume:
    driver: local
```

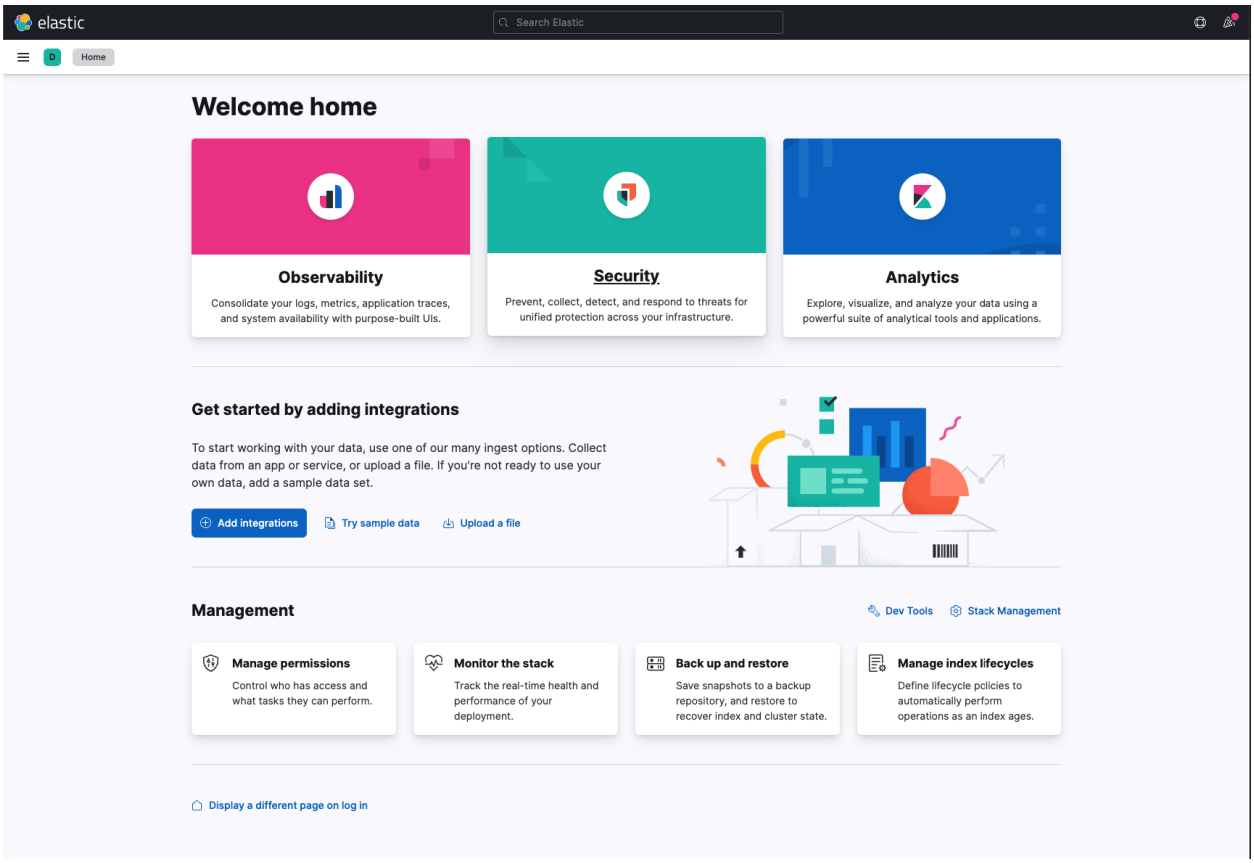
Now when we have defined your docker compose file we can try to run our first Docker compose app. Be sure you are in same folder like your docker-compose file and type ~ `docker compose up -d` into terminal. The `-d` flag means that your app will be running in detached mode. You can check if all containers are running with `docker ps` command.

You should see this:

```
CONTAINER ID   IMAGE                                                 COMMAND                  CREATED             STATUS             PORTS                                            NAMES
c1de59143305   docker.elastic.co/kibana/kibana:8.0.0                 "/bin/tini -- /usr/l…"   About a minute ago   Up About a minute   0.0.0.0:5601->5601/tcp                           kibana
b3508ac999d5   docker.elastic.co/elasticsearch/elasticsearch:8.0.0   "/bin/tini -- /usr/l…"   About a minute ago   Up About a minute   0.0.0.0:9200->9200/tcp, 0.0.0.0:9300->9300/tcp   elasticsearch
```

You can also enter the Kibana GUI. Go to your browser and type `localhost:5601` into search bar. You can see that you type localhost port which we define in the docker compose file.

Wow! If everything is okay you will see this:



Run Weather pump to pump data to Elastic Search index

Well done! We installed ElasticSearch and Kibana locally and we are able to access the ElasticSearch with Kibana GUI. Now we can try to run pump which take weather data and we store them in Elasticsearch index. We already build Weather pump image so you basically pull the image from Docker hub and run it.

To do it simply run this command in your terminal:

```
~ docker run --network=host -dit lukasvecerka/bspump-weather
```

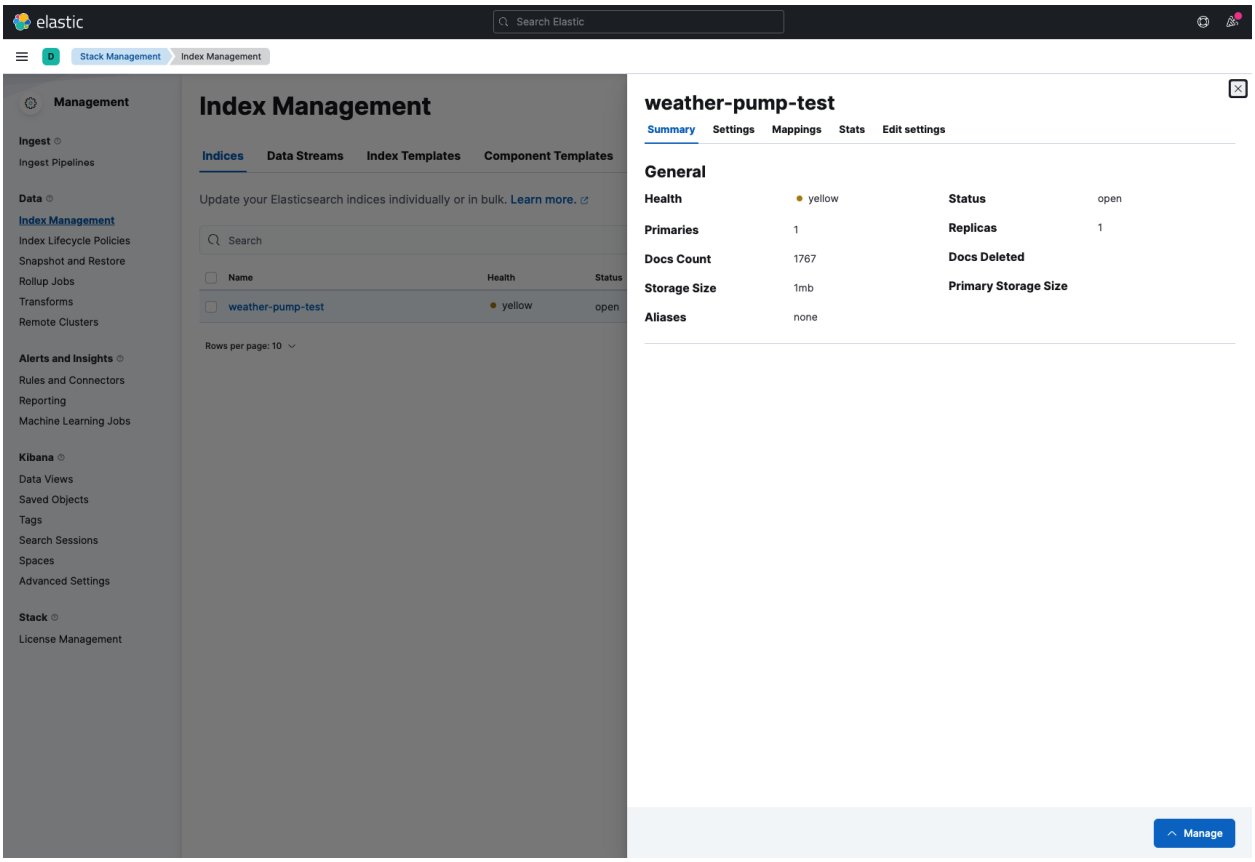
You have to set `--network=host` which mean that your container can now access the localhost on your host machine.

If you type `docker ps` the incoming output in terminal should be this:

```
CONTAINER ID   IMAGE                                                 COMMAND                  CREATED         STATUS         PORTS                                                NAMES
70d2c5fdf018   lukasvecerka/bspump-weather                           "python3 bspump-weat…"   7 seconds ago   Up 2 seconds   80/tcp                                               bspump-weather
28e54d817c93   docker.elastic.co/kibana/kibana:8.0.0                 "/bin/tini -- /usr/l…"   7 seconds ago   Up 4 seconds   0.0.0.0:5601->5601/tcp                               kibana
22fa6b920c89   docker.elastic.co/elasticsearch/elasticsearch:8.0.0   "/bin/tini -- /usr/l…"   8 seconds ago   Up 5 seconds   0.0.0.0:9200->9200/tcp, 0.0.0.0:9300->9300/tcp       elasticsearch
```

Now go to this url address. Its page of Index Management where you can see all of your stored indexes.

If your containers are running correctly you can see that there is index called `weather-pump-test`. This is the index where we store data from our weather pump.



Summarize

That's all for this example! In this example we learnt how to work with Docker and especially with Docker compose tool. How to set services in our application in Docker compose. As conclusion we installed ElasticSearch and Kibana locally and pump data on index in ElasticSearch with our pump.

What next

In the future you can add more services into your docker compose application and extend your environment with this services. You can build your own Docker image and push it to Docker hub and then use it in your docker compose.

More about how to create BSPump Docker image is here *Docker File Quickstart*

2.2.9 Install Kafka and KafDrop via Docker

About

This example is focused on how to install Kafka nad KafDrop on your localhost and search topics from Kafka in KafDrop. We will be using Docker and Docker Compose to install ElasticSearch environment. Be sure you have set up Docker and Docker Compose. If not follow this guide to install Docker and Docker compose.

In the end we will use Docker image of our Coindesk API pump, which can be found here coindesk, to pump data to topic in our local Kafka.

Docker is a platform which provides the ability to package and run an application in a loosely isolated environment called a container. More about Docker.

You can also read our quickstart how to use Docker with BSPump module here: *Docker File Quickstart*

Docker compose is a tool for defining and running multi-container Docker applications. More about Docker compose.

Docker compose with Kafka and KafDrop

Now we create Docker Compose file to run Kafka and KafDrop on our localhost. Create `docker-compose.yml` file in our desired folder. In docker compose you have to define your services which you want to use. Each service is one container which will be running. In our case we define `zookeeper`, `kafka` and `kafdrop`. ZooKeeper is essentially a service for distributed systems offering a hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems.

Services are consist of these values:

1. *image*: we choose which image will be download from DockerHub (after we run the docker compose its automatically pull the image)
2. *hostname*: name of service in multi-container network
3. *ports*: specified ports where the container will runs
4. *environments*: setting up the services configuration (e.g. Kafka Broker ID etc.)
5. *depends_on*: service will wait until specified service in depends_on will start
6. *restart*: service try to restart after unexpected end

Just copy-paste this chung od code into you `docker-compose.yml` file:

```
version: '3.9'
services:
  zookeeper:
    image: zookeeper:3.4.9
    hostname: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOO_MY_ID: 1
      ZOO_PORT: 2181
      ZOO_SERVERS: server.1=zookeeper:2888:3888
    volumes:
      - /data/zookeeper/data:/data
      - /data/zookeeper/datalog:/datalog

  kafka1:
    image: confluentinc/cp-kafka:5.3.0
    hostname: kafka1
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_LISTENERS: LISTENER_DOCKER_INTERNAL://kafka1:19092,LISTENER_
↪DOCKER_EXTERNAL://${DOCKER_HOST_IP:-127.0.0.1}:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: LISTENER_DOCKER_INTERNAL:PLAINTEXT,LISTENER_
↪DOCKER_EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: LISTENER_DOCKER_INTERNAL
```

(continues on next page)

(continued from previous page)

```
    KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
    KAFKA_BROKER_ID: 1
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  volumes:
    - /data/kafka1/data:/var/lib/kafka/data
  depends_on:
    - zookeeper

kafdrop:
  image: obsidiandynamics/kafdrop
  restart: "no"
  ports:
    - "9000:9000"
  environment:
    KAFKA_BROKERCONNECT: "kafka1:19092"
  depends_on:
    - kafka1
```

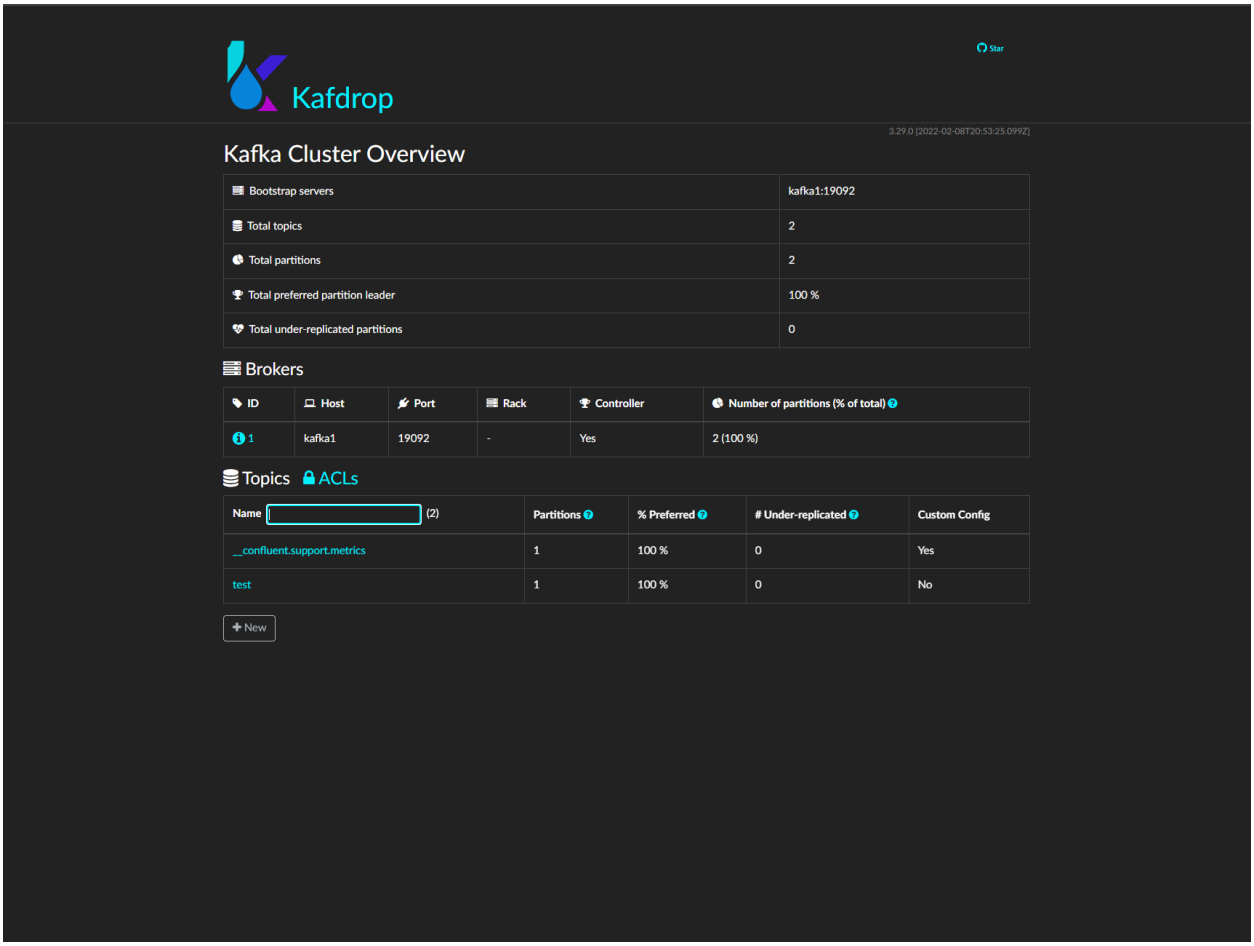
Now when we have defined your docker compose file we can try to run our first Docker compose app. Be sure you are in same folder like your docker-compose file and type ~ `docker compose up -d` into terminal. The `-d` flag means that your app will be running in detached mode. You have to wait little bit when all the images is pulled. After that you can check if all containers are running with `docker ps` command.

You should see this:

```
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS                                            NAMES
f5c54fd91a93   obsidiandynamics/kafdrop       "/kafdrop.sh"            7 seconds ago    Up 3 seconds    0.0.0.0:9000->9000/tcp                           local-host-kafka-kafdrop-1
9a8654045f21   confluentinc/cp-kafka:5.3.0    "/etc/confluent/dock…"   7 seconds ago    Up 4 seconds    0.0.0.0:9092->9092/tcp                           local-host-kafka-kafka1-1
0cfcbff206d0   zookeeper:3.4.9                "/docker-entrypoint.…"   7 seconds ago    Up 5 seconds    2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp       local-host-kafka-zookeeper-1
```

You can also enter the KafDrop. Go to your browser and type `localhost:9000` to the search bar. You can see that you specify the port that we setup in docker compose.

Wow! If everything work correctly you can see thin page:



Pump data to Kafka topic

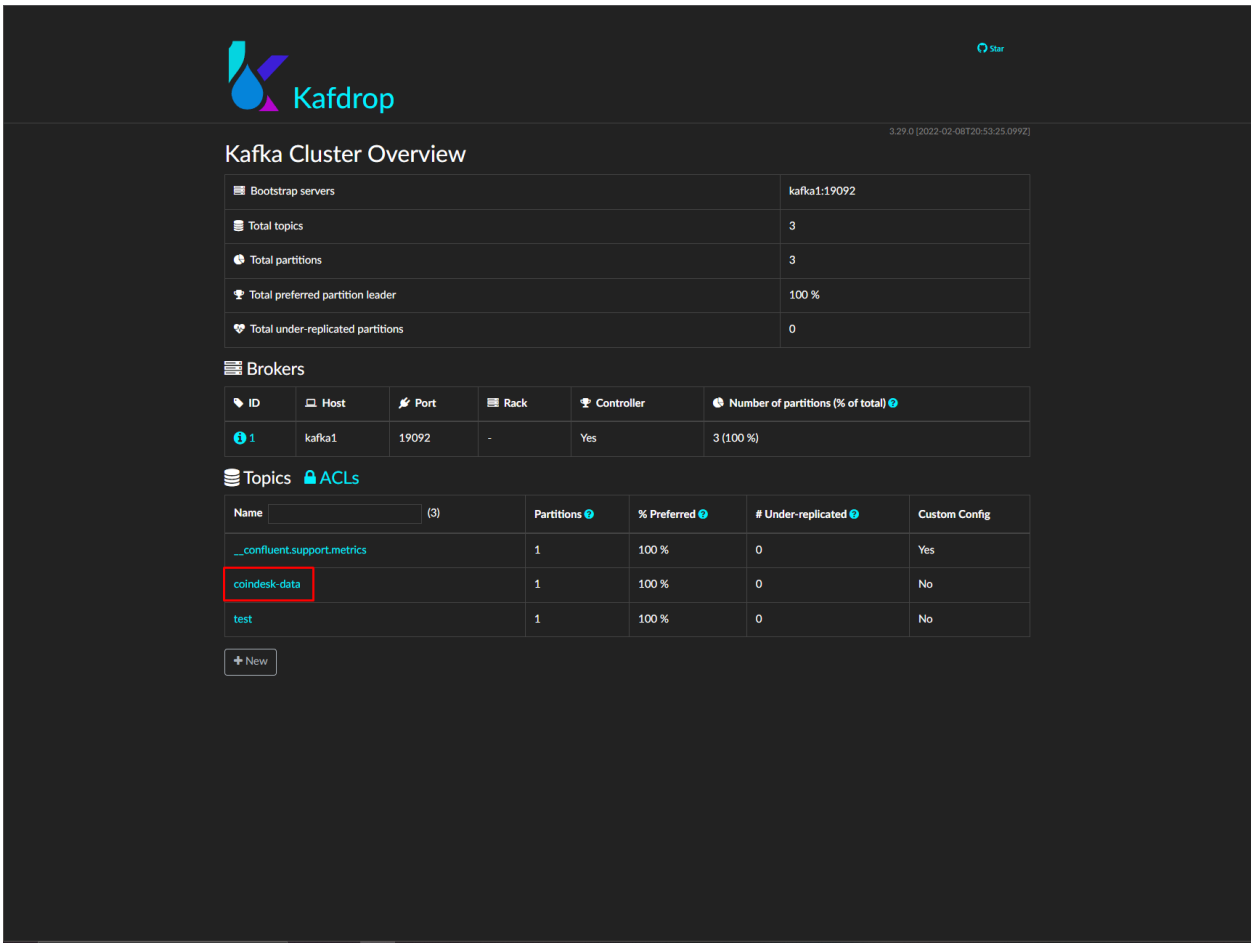
Well done! We've installed Kafka and KafDrop locally and we are able to see topics in KafDrop. Now we can try to run pump which take data from CoinDesk API and store them in Kafka topic. We already build the Coindesk pump image so you basically use the image and run it.

Simply type this command to your terminal and we will see what's happen.

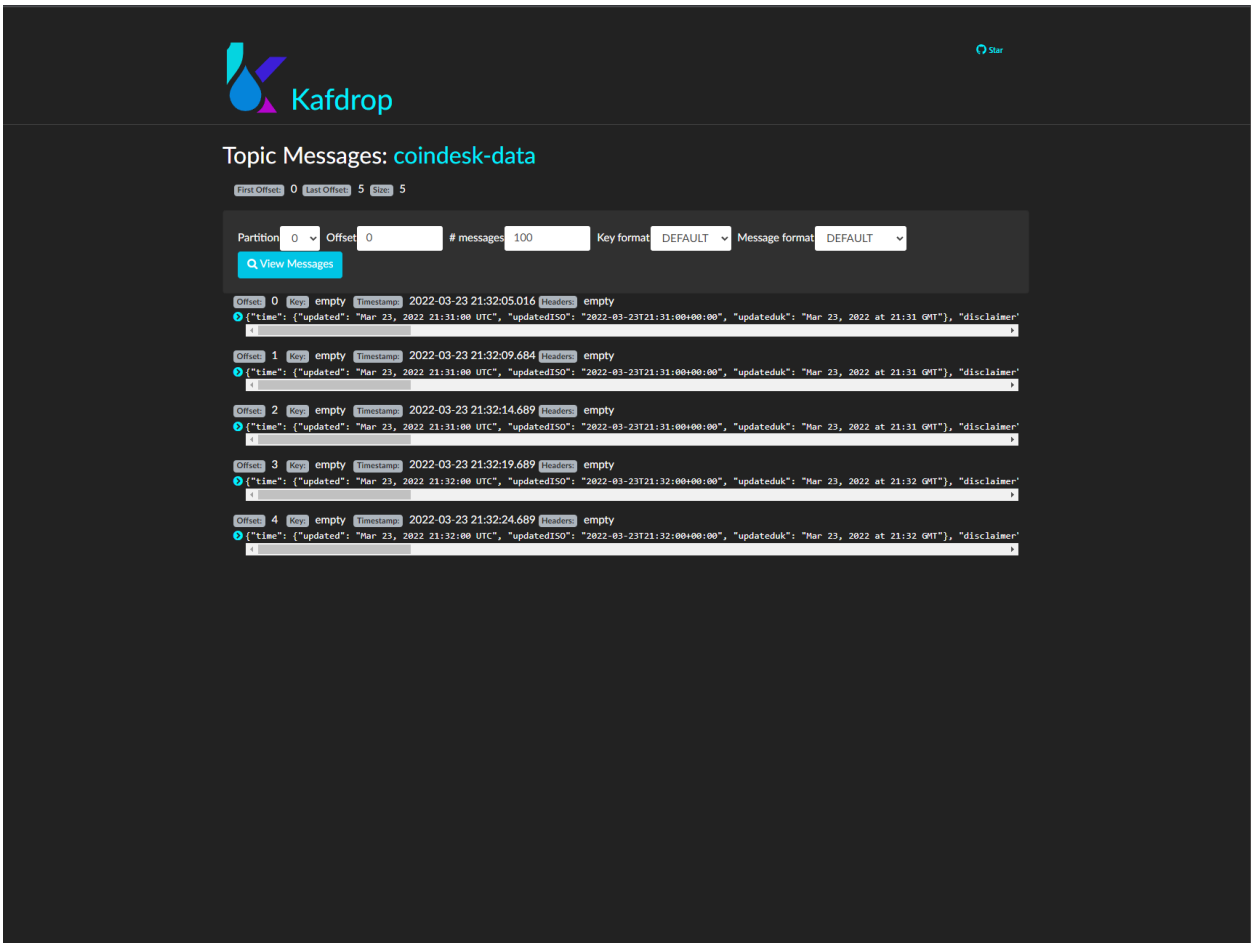
```
~ docker run --network=host -dit lukasvecerka/bspump-kafkasink-example
```

You have to set `--network=host` which mean that your container can now access the localhost on your host machine.

Now when you look into KafDrop you can see `coindesk-data` topic:



You can look on messages which you pump to this topic. Just click on topic name, then on `View Messages` and again on `View Messages` and you should see something like this:



Summarize

That's all for this example! In this example we learnt how to work with Docker and especially with Docker compose tool. We learnt how to set services in our application in Docker compose. In the end we installed Kafka, Zookeeper and KafDrop locally and we run pump with Docker container to pump data to Kafka topic.

What next

In the future you can add more services into your docker compose application nad extend your environment with this services. You can build your own Docker image and push it to Docker hub and then use it in your docker compose or simply run it as a container.

More about how to create BSPump Docker image is here *Docker File Quickstart*

2.2.10 Docker File Quickstart

About

This tutorial will help you to create your own Docker image for your pipeline. First things first, I would recommend you to go through Docker Documentation if this is your first time with Docker.

quickstart to docker

Docker can help you with deployment of your app on other devices. Everything you need to do is to setup docker one device and then it works on every other device. Firstly you have to create docker image for you application. In our case we will create image for our BS Pipeline. To do that we have to firstly create a docker file for our pipeline.

We will be using code from one of our examples coindesk. You can simply copy paste the code and everything should be working if you have a bspump python module installed

docker file

Creating a docker file is very easy thing to do. You have only copy-paste the code below

```
FROM  teskalabs/bspump:nightly

WORKDIR /opt/coindesk

COPY coindesk.py ./coindesk.py

CMD ["python3", "coindesk.py"]
```

To explain what is does:

1. keyword `FROM` specifies what docker image you are using. In this case we will be using a "preset" for a bspump. This image is running on Alpine linux and has all libraries installed.

2. `WORKDIR` specifies the name of your working directory to where other files will be copied

3. `COPY` this command is used to copy any files you will be using including the source code of your app.

4. `CMD` is a command for running commands in your container. You have to write a command sequence as a list where each element is one word of the command. In our case we want to execute our program using `python3 coindesk.py`

Creating docker image

To build your docker image use this command. Make sure to use -t switch and match <<your docker nickname>> to your docker login name. This must match for successful push of the image to the docker desktop.

```
docker build -t <<your docker nickname>>/<<name of your image>> .
```

Now you can try to run your docker image using:

```
docker run -it <<your docker nickname>>/<<name of your image>>
```

now your container should be running in your console. If you want to terminate it open another console and type

```
docker ps
```

This command will show you all your running containers and with

```
docker kill <<CONTAINER ID>>
```

It will terminate the container. Container ID should be found next to the running image after typing `docker ps`

If you want to see all containers that were initiated type

```
docker ps -a
```

Now if you want to use this image from other devices for docker compose for example. You can push the image to your repository using:

```
docker push <<your docker nickname>>/<<name of your image>>
```

if you haven't tagged your container before use

```
docker tag <<name of your image>> <<your docker nickname>>/<<name of your image>>
```

Now you should have running docker container and you know how to push it to your docker hub. If you are still not sure how to use docker I would recommend to check docker documentation once again. Docker is not complicated, but it takes some time to get used to it.

additional commands

TODO

what next

if you have successfully created your own docker image you can try to connected your pipeline with other technologies like elastic search or kafka. Check our *Install ElasticSearch and Kibana via Docker* for working with docker compose.

2.2.11 WebSocket Example

This example will show you how can you can connect two pipelines connection using socket server connection.

what is socket

Socket is a peer-to-peer connection between two computers. You can imagine it like two computers have access to one directory and can share data between each other.

explain server/client consumer/producer

The pipeline you will create can be either a server or a client. Server is a script that listens on a certain IP address and port, client is the one who "connects" to a certain port of the server. Both client and server can be either consumers, meaning that consumer (consumes) the data, and producer is the one who produce the data. The specific combination of server/client consumer/producers mainly depends on what do you wanna do. In this example we will show both server/consumer - client/producer type of connection and server/producer - client/consumer connection.

Server consumer

Server consumer means that this pipeline will be waiting for any client trying to make a connection and if there is a connection with a client the server will get the incoming data into its pipeline. This server pipeline will use Websocket Source as its Source.

To create this kind of pipeline we have to use our WebSocketSource and specify the address and port on which it will listen for any possible connections. In this example we will run both pipelines on localhost, so you do not have to waste your time setting up your own network.

```
#!/usr/bin/env python3
import bspump
import bspump.common
import bspump.web
import bspump.http


class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)



        self.build(
            bspump.http.WebSocketSource(app, self),
            bspump.common.PPrintSink(app, self)
        )


if __name__ == '__main__':
    app = bspump.BSPumpApplication(web_listen="0.0.0.0:8080") #set web_listen variable␣
↪to the address you want

    svc = app.get_service("bspump.PumpService")

    # Construct and register Pipeline
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)

    #you have to use add_get method to set up address using the handler.
    app.WebContainer.WebApp.router.add_get('/bspump/ws', pl.WebSocketSource.handler)

    app.run()
```

You can copy-paste the code above. The pipeline is really simple the only thing you have to do is to add WebSocket Source. Just make sure to set up the `web_listen` variable in the `BSPumpApplication` object, and do not forget that you have to call the `add_get` method **TODO**

Now you can run the script and your server should be running listening for any possible connections.

Client producer

We have a running server, so now we have to create a client that can connect to the server and feed it with the data.

```
#!/usr/bin/env python3
import bspump
import bspump.common
import bspump.http
import bspump.trigger


class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)
        self.Counter = 1

        # self.Source = bspump.common.InternalSource(app, self)

        self.build(
            bspump.http.HTTPClientSource(app, self, config={
                'url': 'https://api.coindesk.com/v1/bpi/currentprice.json'
                # Trigger that triggers the source every second (based on the method␣
↪parameter)
            }).on(bspump.trigger.PeriodicTrigger(app, 5)),

            bspump.http.HTTPClientWebSocketSink(app, self, config={
                'url': 'http://127.0.0.1:8080/bspump/ws',
            })

        )

if __name__ == '__main__':
    app = bspump.BSPumpApplication()

    svc = app.get_service("bspump.PumpService")

    # Construct and register Pipeline
    pl = SamplePipeline(app, 'SamplePipeline')
    svc.add_pipeline(pl)

    app.run()
```

Creating the client is much more easier than the server. All you have to do is to use `HTTPClientSocketSink` with config where you specify the url of the server you want to connect to. In this case it is `http://127.0.0.1:8080/bspump/ws`

what next

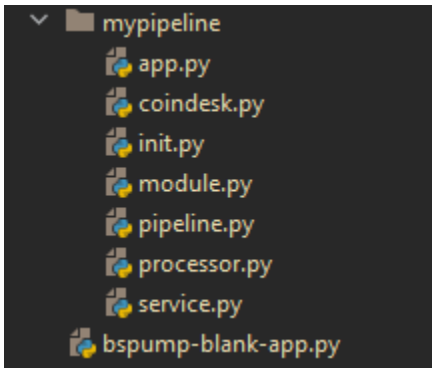
This example should have you given an idea how to use and connect pipelines using socket connection.

2.2.12 Blank App

In this tutorial you will learn how divide a pipeline into several file components. This approach is beneficial for creating more advanced pipelines as you can use some of the components without the need of copy pasting code. It is also much more clear. This is a general guide so you can apply this structure to your pipeline. We will be using so-called blank app in this tutorial for simplicity you can find the code here.

In this tutorial we will use code from our previous tutorial coindesk, but don't worry once you create this structure it is easy to make changes for your own pipeline.

first you will create similar file hierarchy like on this image.



pipeline

In this file you will have your pipelien with `self.build` method. If you want to use your own processors, sources or sinks you have to import them from another file. In this example I want to use my processor for coindesk, so I have to use

```
from .processor import EnrichProcessor
```

and then I can reference it in `self.build` method.

```
import bspump
import bspump.common
import bspump.http
import bspump.trigger


from .processor import EnrichProcessor


class SamplePipeline(bspump.Pipeline):

    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
```

(continues on next page)

(continued from previous page)

```
            # Source that GET requests from the API source.
            bspump.http.HTTPClientSource(app, self, config={
                'url': 'https://api.coindesk.com/v1/bpi/currentprice.json'
                # Trigger that triggers the source every second (based on the method␣
↪parameter)
            }).on(bspump.trigger.PeriodicTrigger(app, 5)),
            # Converts incoming json event to dict data type.
            bspump.common.StdJsonToDictParser(app, self),
            # Adds a CZK currency to the dict
            EnrichProcessor(app, self),
            bspump.common.StdDictToJsonParser(app,self),
            # prints the event to a console
            bspump.common.PPrintSink(app, self),
        )
```

Only remember that name of your pipeline (the name of the class) will be used in other files.

processor

To create processor file you can simply copy-paste your processor class.

note
Do not forget to import bspump module, so your processor can function normally.

```
import bspump


class EnrichProcessor(bspump.Processor):
    def __init__(self, app, pipeline, id=None, config=None):
        super().__init__(app, pipeline, id=None, config=None)

    def convertUSDtoJPY(self, usd):
        return usd * 113.70  # outdated rate usd/jpy

    def process(self, context, event):
        jpyPrice = str(self.convertUSDtoJPY(event["bpi"]["USD"]["rate_float"]))

        event["bpi"]["JPY"] = {
            "code": "JPY",
            "symbol": "&yen;",
            "rate": ''.join((jpyPrice[:3], ',', jpyPrice[3:])),
            "description": "JPY",
            "rate_float": jpyPrice
        }

        return event
```

service

In service you have to register your pipeline. You can also register more pipelines.

note
Remember to import your pipeline class here, so you can register the pipeline.

```
import asab

from .pipeline import SamplePipeline


class BlankService(asab.Service):

    def __init__(self, app, service_name="blank.BlankService"):
        super().__init__(app, service_name)

    async def initialize(self, app):
        svc = app.get_service("bspump.PumpService")

        # Create and register all connections here

        # Create and register all matrices here

        # Create and register all lookups here

        # Create and register all pipelines here

        self.SamplePipeline = SamplePipeline(app, "SamplePipeline")
        svc.add_pipeline(self.SamplePipeline)

        await svc.initialize(app)
```

```
self.SamplePipeline = SamplePipeline(app, "SamplePipeline")
svc.add_pipeline(self.SamplePipeline)
```

These two lines of the code register your pipeline.

module

In module you create a module of your service. You can create more modules from several services.

```
import asab

from .service import BlankService


class BlankModule(asab.Module):
    def __init__(self, app):
        super().__init__(app)

        self.BlankService = BlankService(app)
```

app

In app you create the whole application. You have to only include the module you have created. You can include more modules here.

```
import bspump


class BlankAppApplication(bspump.BSPumpApplication):

    def __init__(self):
        super().__init__()

        from .module import BlankModule
        self.add_module(BlankModule)
```

init

create this file for initialization of your pipeline.

```
from .app import BlankAppApplication
```

how to start the pipeline

to start your pipeline create another file. For example, `bspump-blank-app.py` and copy-paste this code

```
from mypipeline.app import BlankAppApplication

if __name__ == '__main__':
    app = BlankAppApplication()
    app.run()
```

`mypipeline.app` is the path to your `app` python file. and `BlankAppApplication` is the name of your pipeline class.

Then you create an object of your class and run it.

2.3 Reference Documentation

BSPump Reference Documentation describes the *bspump* Python library. Based on ASAB library. ASAB is a platform that enables BSPump to be efficient and easy to configure.

2.3.1 Basics

Basics covers the most fundamental components of a BSPump. We will start with the "backbone" of the BSPump, which is called a "pipeline".

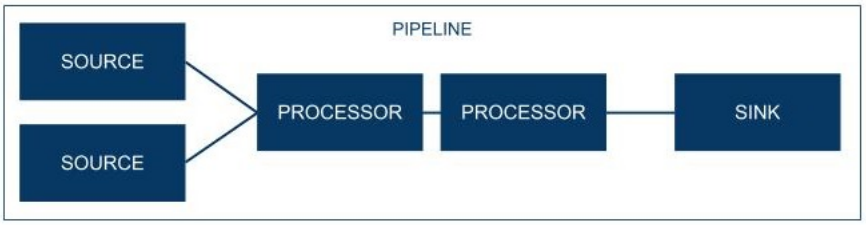
Pipeline

The pipeline class is responsible for construction of the BSPump pipeline itself. Its methods enable us to maintain a working lifecycle of the system.

Pipeline is responsible for **data processing** in BSPump. Individual *Pipeline* objects work **asynchronously** and **independently** of one another (provided dependence is not defined explicitly – for instance on a message source from some other pipeline) and can be triggered in unlimited numbers. Each *Pipeline* is usually in charge of **one** concrete task.

Pipeline has three main components:

- Source
- *Processor*
- *Sink*



BitSwan PIPELINE

Source connects different **data sources** with the *Pipeline* to be processed

Multiple sources

A *Pipeline* can have multiple sources. They are simply passed as a list of sources to a *Pipeline* *build()* method.

```
class MyPipeline(bspump.Pipeline):

   def __init__(self, app, pipeline_id):
      super().__init__(app, pipeline_id)
      self.build(
         [
            MySource1(app, self),
            MySource2(app, self),
            MySource3(app, self),
         ]
         bspump.common.NullSink(app, self),
      )
:meta private:
```

The main component of the BSPump architecture is a so-called *Processor*. This object **modifies**, **transforms** and **enriches** events. Moreover, it is capable of **calculating metrics** and **creating aggregations**, **detecting anomalies** or react to known as well as unknown **system behaviour patterns**.

Processors differ as to their **functions** and all of them are aligned according to a predefined sequence in **pipeline objects**. As regards working with data events, each *Pipeline* has its unique task.

Processors are passed as a **list** of *Processors* to a *Pipeline* *build()* method

```
class MyPipeline(bspump.Pipeline):

   def __init__(self, app, pipeline_id):
      super().__init__(app, pipeline_id)
      self.build(
         [
            MyProcessor1(app, self),
            MyProcessor2(app, self),
            MyProcessor3(app, self),
         ]
         bspump.common.NullSink(app, self),
      )
:meta private:
```

Sink object serves as a **final event destination** within the pipeline given. Subsequently, the event is dispatched/written into the system by the BSPump.

class Pipeline(*app*, *id=None*, *config=None*)

Bases: `ABC`, `Configurable`

Description: Pipeline is ...

An example of The *Pipeline* construction:

```
class MyPipeline(bspump.Pipeline):

       def __init__(self, app, pipeline_id):
              super().__init__(app, pipeline_id)
              self.build(
                     [
                        MySource(app, self),
                        MyProcessor(app, self),
                        MyProcessor2(app, self),
                     ]
                     bspump.common.NullSink(app, self),
              )
```

Pipeline construction

The following are the core methods of the pipeline.

`Pipeline.build(source, *processors)`

This method enables to add sources, *Processors*, and sink to create the structure of the *Pipeline*.

Parameters

source
[str] ID of a source.

***processors**
[str, list optional] ID of *Processor* or list of IDs.

`Pipeline.set_source(source)`

Sets a specific source or list of sources to the *Pipeline*.

Parameters

source
[str, list optional] ID of a source.

If a list of sources is passed to the method, it adds the entire list of sources to the *Pipeline*.

`Pipeline.append_processor(processor)`

Adds a *Processors* to the current *Pipeline*.

Parameters

processor
[str] ID of a *processor*.

Hint
The Generator can be added by using this method. It requires a depth parameter.

`Pipeline.remove_processor(processor_id)`

Removes a specific *processor* from the *Pipeline*.

Parameters

processor_id
[str] ID of a *processor*.

Returns
Error when *processor* is not found.

`Pipeline.insert_before(id, processor)`

Inserts the *Processor* into the *Pipeline* in front of another *processor* specified by ID.

Parameters

id
[str] ID of a *processor* that we want to insert.

processor
[str] Name of the *processor* in front of which will be inserted the new *processor*.

Returns
True on success. False if ID was not found.

Pipeline.**insert_after**(*id*, *processor*)

Inserts the *Processor* into the *Pipeline* behind another *Processors* specified by ID.

Parameters

id
[str] ID of a processor that we want to insert.

processor
[str] Name of a *processor* after which we insert our *processor*.

Returns
True if successful. False if ID was not found.

Pipeline.**iter_processors**()

Uses python generator routine that iterates through all *Processors* in the *Pipeline*.

Yields
A Processor from a list in the *Pipeline*.

Other Pipeline Methods

The additional methods below bring more features to the pipeline. However, many of them are very important and almost necessary.

Pipeline.**build**(*source*, **processors*)

This method enables to add sources, *Processors*, and sink to create the structure of the *Pipeline*.

Parameters

source
[str] ID of a source.

***processors**
[str, list optional] ID of *Processor* or list of IDs.

Pipeline.**iter_processors**()

Uses python generator routine that iterates through all *Processors* in the *Pipeline*.

Yields
A Processor from a list in the *Pipeline*.

Other pipeline methods

Pipeline.**time**()

Returns correct time.

Returns
App.time()

Hint
More information in the ASAB documentation in UTC Time.

Pipeline.**get_throttles**()

Returns components from *Pipeline* that are throttled.

Returns
self._throttles Return list of throttles.

Pipeline.**is_error**()

Returns False when there is no error, otherwise it returns True.

Returns
self._error is not None.

Pipeline.**set_error**(*context*, *event*, *exc*)

When called with *exc is None*, it resets error (aka recovery).

When called with exc, it sets exceptions for soft errors.

Parameters

context
[type?] Context of an error.

event
[Data with time stamp stored in any data type usually is in JSON.] You can specify an event that is passed to the method.

exc
[Exception.] Python default exceptions.

Pipeline.**handle_error**(*exception*, *context*, *event*)

Used for setting up exceptions and conditions for errors. You can implement it to evaluate processing errors.

Parameters

exception
[Exception] Used for setting up a custom Exception.

context
[information] Additional information can be passed.

event
[Data with time stamp stored in any data type, usually it is in JSON.] You can specify an event that is passed to the method.

Returns
False for hard errors (stop the *Pipeline* processing). True for soft errors that will be ignored.

Example:

```
class SampleInternalPipeline(bspump.Pipeline):

        def __init__(self, app, pipeline_id):
                super().__init__(app, pipeline_id)

                self.build(
                        bspump.common.InternalSource(app, self),
                        bspump.common.JSONParserProcessor(app, self),
                        bspump.common.PPrintSink(app, self)
                )

        def handle_error(self, exception, context, event):
                if isinstance(exception, json.decoder.JSONDecodeError):
                        return True
                return False
```

Pipeline.**link**(*ancestral_pipeline*)

Links this *Pipeline* with an ancestral *Pipeline*. This is needed e. g. for a propagation of the throttling from child *Pipelines* back to their ancestors. If the child *Pipeline* uses InternalSource, it may become throttled because the internal queue is full. If so, the throttling is propagated to the ancestral *Pipeline*, so that its source may block incoming events until the internal queue is empty again.

Parameters

ancestral_pipeline
[str] ID of a *Pipeline* that will be linked.

Pipeline.**unlink**(*ancestral_pipeline*)

Unlinks an ancestral *Pipeline* from this *Pipeline*.

Parameters

ancestral_pipeline
[str] ID of a ancestral *Pipeline* that will be unlinked.

Pipeline.**throttle**(*who*, *enable=True*)

Enables throttling method for a chosen *pipeline* and its ancestral *pipelines*,x if needed.

Parameters

who
[ID of a *processor*.] Name of a *processor* that we want to throttle.

enable
[bool, defualt True] When True, content of argument 'who' is added to _throttles list.

async Pipeline.**ready**()

Checks if the *Pipeline* is ready. The method can be used in source: *await self.Pipeline.ready()*.

Pipeline.**is_ready**()

This method is a check up of the event in the Event class.

Returns
_ready.is_set().

Pipeline.**inject**(*context*, *event*, *depth*)

Injects method serves to inject events into the *Pipeline*'s depth defined by the depth attribute. Every depth is interconnected with a generator object.

Parameters

context
[string] Information propagated through the *Pipeline*.

event
[Data with time stamp stored in any data type, usually it is in JSON.] You can specify an event that is passed to the method.

depth
[int] Level of depth.

Note
For normal operations, it is highly recommended to use process method instead.

async Pipeline.**process**(*event*, *context=None*)

Process method serves to inject events into the *Pipeline*'s depth 0, while incrementing the event in metric.

Parameters

event
[Data with time stamp stored in any data type, usually it is in JSON.] You can specify an event that is passed to the method.

context
[str, default None] You can add additional information needed for work with event streaming.

Hint
This is recommended way of inserting events into a *Pipeline*.

Pipeline.**create_eps_counter**()

Creates a dictionary with information about the *Pipeline*. It contains eps (events per second), warnings and errors.

Returns
self.MetricsService Creates eps counter using MetricsService.

Note
EPS counter can be created using this method or dicertly by using MatricsService method.

Pipeline.**ensure_future**(*coro*)

You can use this method to schedule a future task that will be executed in a context of the *Pipeline*. The *Pipeline* also manages a whole lifecycle of the future/task, which means, it will collect the future result, trash it, and mainly it will capture any possible exception, which will then block the *Pipeline* via set_error().

Parameters

coro
[??] ??

Hint
If the number of futures exceeds the configured limit, the *Pipeline* is throttled.

Pipeline.**locate_source**(*address*)

Locates a sources based on its ID.

Parameters

address
[str] ID of the source.

Pipeline.**locate_connection**(*app*, *connection_id*)

Finds a connection by ID.

Parameters

app
[Application] Name of the Application.

connection_id
[str] ID of connection we want to locate.

Returns
connection

Pipeline.**locate_processor**(*processor_id*)

Finds a *Processor* by ID.

Parameters

processor_id
[str] ID of a *Processor*.

Returns
processor

Pipeline.**start**()

Starts the lifecycle of the *Pipeline*.

async Pipeline.**stop**()

Gracefully stops the lifecycle of the *Pipeline*.

Source

class Source(*app*, *pipeline*, *id=None*, *config=None*)

Bases: Configurable

Source class is responsible for connecting to a source, and propagating events or other data from the source to the *processors*.

Source.**__init__**()

Set the initial ID, *Pipeline* and Task.

Parameters

app
[Application] Name of an *Application* <https://asab.readthedocs.io/en/latest/asab/application.html#>`_ .

pipeline
[address of a pipeline] Name of a *Pipeline*.

id
[str, default None] Name of a the *Pipeline*.

config
[compatible config type , default None] Option for adding a configuration file.

2.3.2 Source Construction

Source is an **object** designed to obtain data from a predefined input. The BSPump contains a lot of universally usable, specific source objects, which are capable of loading data from known data interfaces. The BitSwan product further expands these objects by adding source objects directly usable for specific cases of use in industry field given.

Each source represent a coroutine/Future/Task that is running in the context of the main loop. The coroutine method `main()` contains an implementation of each particular source.

Source MUST await a *Pipeline* ready state prior producing the event. It is acomplished by *await self.Pipeline.ready()* call.

class Source(*app*, *pipeline*, *id=None*, *config=None*)

Bases: `Configurable`

Source class is responsible for connecting to a source, and propagating events or other data from the source to the *processors*.

__init__(*app*, *pipeline*, *id=None*, *config=None*)

Set the initial ID, *Pipeline* and Task.

Parameters

app
[Application] Name of an *Application* <https://asab.readthedocs.io/en/latest/asab/application.html#>`_ .

pipeline
[address of a pipeline] Name of a *Pipeline*.

id
[str, default None] Name of a the *Pipeline*.

config
[compatible config type , default None] Option for adding a configuration file.

async Source.**process**(*event*, *context=None*)

This method is used to emit event into a *Pipeline*.

Parameters

event: Data with time stamp stored in any data type, usually JSON.
Message or information that is passed to the method and emitted into a *Pipeline*.

context
[default None] Additional information.

If there is an error in the processing of the event, the *Pipeline* is throttled by setting the error and the exception raised.

:hint The source should catch this exception and fail gracefully.

Source.**start**(*loop*)

Starts the *Pipeline* through the _main method, but if main method is implemented it starts the coroutine using main method instead.

Parameters

loop
[?] Contains the coroutines.

async Source.**stop**()

Stops the Source using self.Task. If the processes are not done it cancels them or raises an error.

Source.**restart**(*loop*)

Restarts the loop of coroutines and returns result() method.

Parameters

loop
[??] Contains the coroutines.

async Source.**main**()

Can be implemented for additional features, else will raise NotImplementedError and _main is called instead.

async Source.**stopped**()

Waits for all asynchronous tasks to be completed. It is helper that simplifies the implementation of sources.

Example:

..code:: python

async def main(self):

#... initialize resources here

await self.stopped()

#... finalize resources here

Source.**locate_address**()

Locates address of a *Pipeline*.

Returns
ID and ID of a *Pipeline* as a string.

classmethod Source.**construct**(*app*, *pipeline*, *definition: dict*)

Can create a source based on a specific definition. For example, a JSON file.

Parameters

app
[Application] Name of the Application.

pipeline
[*Pipeline*] Specification of a *Pipeline*.

definition
[dict] Definition that is used to create a source.

Returns
cls(app, newid, config)

This is an abstract source class intended as a base for implementation of 'cyclic' sources such as file readers, SQL extractors etc. You need to provide a trigger class and implement *cycle()* method.

Trigger source will stop execution, when a *Pipeline* is cancelled (raises concurrent.futures.CancelledError). This typically happens when a program wants to quit in reaction to a on the signal.

You also may overload the `main()` method to provide additional parameters for a *cycle()* method.

```
async def main(self):
        async with aiohttp.ClientSession(loop=self.Loop) as session:
                await super().main(session)


async def cycle(self, session):
        session.get(...)
```

class TriggerSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Source*

Description:

Returns

__init__(*app*, *pipeline*, *id=None*, *config=None*)

Set the initial ID, *Pipeline* and Task.

Parameters

app
[Application] Name of an *Application* <https://asab.readthedocs.io/en/latest/asab/application.html#>`_ .

pipeline
[address of a pipeline] Name of a *Pipeline*.

id
[str, default None] Name of a the *Pipeline*.

config
[compatible config type , default None] Option for adding a configuration file.

TriggerSource.**time**()

Method used for measuring an accurate time.

Returns
App.time()

Hint
You can find more information about UTC Time in the ASAB documentation

TriggerSource.**on**()

Sets a Trigger which is a method that waits for a given condition.

Parameters

trigger
[keyword of a trigger] Given condition that.

Returns
Trigger.add(trigger)

async TriggerSource.**main**()

Waits for *Pipeline*, triggers, and calls exceptions when the source is initiated.

Parameters

*args : ?

**kwags : ?

async TriggerSource.**cycle**()

Not implemented.

Parameters

*args : ?

**kwags : ?

TriggerSource.**rest_get**()

Description:

Returns

2.3.3 Processor

The main component of the BSPump architecture is a so called processor. This object modifies, transforms and enriches events. Moreover, it is capable of calculating metrics and creating aggregations, detecting anomalies or react to known as well as unknown system behavior patterns.

Processors differ as to their functions and all of them are aligned according to a predefined sequence in pipeline objects. As regards working with data events, each pipeline has its own unique task.

class Processor(*app*, *pipeline*, *id=None*, *config=None*)

Bases: ProcessorBase

Inherits from ProcessorBase.

__init__(*app*, *pipeline*, *id=None*, *config=None*)

Initializes the Parameters

Parameters

app
[object] Application object.

pipeline
[*Pipeline*] Name of the *Pipeline*.

id
[str, default=None,] ID of the class of config.

config
[JSON, or other compatible formats, default=None] Configuration file.

Processor.**time**()

Accurate representation of a time in the *Pipeline*.

Returns
App.time()

classmethod Processor.**construct**()

Can construct a *processor* based on a specific definition. For example, a JSON file.

Parameters

app
[Application] Name of the *Application <https://asab.readthedocs.io/en/latest/asab/application.html#>_*.

pipeline
[str] Name of the *Pipeline*.

definition
[dict] Set of instructions based on which *processor* can be constructed.

Returns
cls(app, pipeline, id=newid, config=config)

Processor.**process**()

Can be implemented to return event based on a given logic.

Parameters

context :
Additional information passed to the method.

event
[Data with time stamp stored in any data type, usually it is in JSON.] You can specify an event that is passed to the method.

Processor.**locate_address**()

Returns an ID of a *processor* and a *Pipeline*.

Returns
ID of the *Pipeline* and self.Id.

Processor.**rest_get**()

Description:

Returns

Processor.**__repr__**()

Return repr(self).

Sink

Sink object serves as a final event destination within the pipeline given. Subsequently, the event is dispatched/written into the system by the BSPump.

class Sink(*app*, *pipeline*, *id=None*, *config=None*)

Bases: ProcessorBase

Sink is basically a processor. It takes an event sends it to a database where it is stored.

__init__(*app*, *pipeline*, *id=None*, *config=None*)

Initializes the Parameters

Parameters

app
[object] Application object.

pipeline
[*Pipeline*] Name of the *Pipeline*.

id
[str, default=None,] ID of the class of config.

config
[JSON, or other compatible formats, default=None] Configuration file.

Connection

class Connection(*app*, *id=None*, *config=None*)

Bases: ABC, Configurable

Connection class is responsible for creating a connection between items or services within the infrastructure of BSPump. Their main use is to create connection with the main components of BSPump: source, *processor* and sink.

__init__(*app*, *id=None*, *config=None*)

Description:

Parameters

app
[Application] Specification of an Application.

id : default None

config
[JSON or other compatible format, default None] It contains important information and data responsible for creating a connection.

Connection construction

Connection.**time**()

Returns accurate time of the asynchronous process.

Hint

More information in the ASAB documentation in UTC Time.

2.3.4 Top Level Objects

BSPumpApplication

class BSPumpApplication(**args*, ***kwargs*)

Bases: Application

BSPumpApplication is responsible for the main life cycle of the Application. It is based on ASAB Application class

BSPumpApplication.**__init__**()

Initiates the Application and looks for config with additional arguments.

Parameters

args : default= None

web_listen : default= None

BSPumpApplication Construction

BSPumpApplication.**create_argument_parser**()

Enables to create arguments that can be called within the command prompt when starting the application

Returns

parser

BSPumpApplication.**parse_arguments**(*args=None*)

Parses argument in the ASAB Application using super() method.

Parameters

args : default= None

Returns

args

async BSPumpApplication.**main**()

Prints a message about how many pipelines are ready.

BSPumpService

class BSPumpService(*app*, *service_name='bspump.PumpService'*)

Bases: Service

Service registry based on Service object. Read more in ASAB documentation **`Service <https://asab.readthedocs.io/en/latest/asab/service.html`_**.

BSPumpService.**__init__**()

Initializes parameters passed to the Service class.

Parameters

app
[Application] Name of the Application.

service_name
[str, Service name] string variable containing “”bspump.PumpService

BSPumpService Methods

BSPumpService.**locate**(*address*)

locates pipeline, source or processor based on the adressed parameter

Parameters

address
[str, ID] Address of an pipeline component.

BSPumpService.**add_pipeline**(*pipeline*)

Adds a pipeline to the BSPump.

Parameters

pipeline
[Pipeline] Name of the Pipeline.

BSPumpService.**add_pipelines**(**pipelines*)

Adds a pipelines the BSPump.

Parameters

*pipelines
[list] List of pipelines that are add to the BSPump.

BSPumpService.del_pipeline(*pipeline*)

Deletes a pipeline from a list of Pipelines.

**Parameters*

pipeline
[str, ID] ID of a pipeline.

BSPumpService.add_connection(*connection*)

Adds a connection to the Connection dictionary.

Parameters

connection
[str, ID] ID of a connection.

Returns
connection

BSPumpService.add_connections(**connections*)

Adds a connections to the Connection dictionary.

Parameters

*connection
[str, ID] list of IDs of a connections.

BSPumpService.locate_connection(*connection_id*)

Locates connection based on connection ID.

Parameters

connection_id
[ID] Connection ID.

BSPumpService.add_lookup(*lookup*)

Sets a lookup based on Lookup.

Parameters

lookup
[Lookup] Name of the Lookup.

Returns
lookup

BSPumpService.add_lookups(**lookups*)

Adds a list of lookups to the Pipeline.

Parameters

lookup
[Lookup] List of Lookups.

BSPumpService.locate_lookup(*lookup_id*, *context=None*)

Locates lookup based on ID.

Parameters

lookup_id
[ID] ID of a Lookup.

context
[,default = None] Additional information.

Returns
lookup from the lookup service or form the internal dictionary.

`BSPumpService.add_lookup_factory(lookup_factory)`

Adds a lookup factory

Parameters

lookup_factory :
Name of lookup factory.

`BSPumpService.add_matrix(matrix)`

Adds a matrix to the Pipeline.

Parameters

matrix
[Matrix] Name of Matrix.

Returns
matrix

`BSPumpService.add_matrixes(*matrixes)`

Adds a list of Matrices to the Pipeline.

Parameters

***matrixes**
[list] List of matrices.

`BSPumpService.locate_matrix(matrix_id)`

Locates a matrix based on matrix ID

Parameters

matrix_id
[str, ID] ID of a matrix.

`async BSPumpService.initialize(app)`

Initializes an Application based on ASAB Application

Parameters

app
[Application] Name of the Application

`async BSPumpService.finalize(app)`

Stops all the pipelines

Parameters

app
[Application] Name of the Application

2.3.5 Common

Aggregator

Aggregation Strategy

class AggregationStrategy

Bases: ABC

Aggregation Strategy

AggregationStrategy.**__init__**()

Aggregation Strategy Methods

abstract AggregationStrategy.**append**(*context*, *event*)

Appends

Parameters

context :

event :

abstract AggregationStrategy.**flush**()

Flushes

abstract AggregationStrategy.**is_empty**() → bool

Description:

List Aggregation Strategy

class ListAggregationStrategy

Bases: *AggregationStrategy*

Description: ... test

ListAggregationStrategy.__init__()

Description:

List Aggregation Strategy Methods

ListAggregationStrategy.append(*context*, *event*)

Description:

Parameters

context :

event :

ListAggregationStrategy.flush()

Description:

Returns

result

ListAggregationStrategy.is_empty() → bool

Description:

Returns

Aggregated Event

String Aggregation Strategy

class StringAggregationStrategy(*delimiter='\n'*)

Bases: *AggregationStrategy*

Description:

StringAggregationStrategy.__init__()

Description:

String Aggregation Strategy Methods

StringAggregationStrategy.**append**(*context*, *event*)

Description:

Parameters

context :

event
[Data with time stamp stored in any data type usually is in JSON.] You can specify an event that is passed to the method.

StringAggregationStrategy.**flush**()

Description:

Returns
result

StringAggregationStrategy.**is_empty**() → bool

Description:

Returns
Aggregated event

Aggregator

class Aggregator(*app*, *pipeline*, *aggregation_strategy: ~bspump.common.aggregator.AggregationStrategy = <bspump.common.aggregator.ListAggregationStrategy object>*, *id=None*, *config=None*)

Bases: *Generator*

Description:

Aggregator.**__init__**()

Description:

Aggregator

`Aggregator.`**`flush`**`()`

Description:

Returns

??

`Aggregator.`**`process`**`(`*context*, *event*`)`

Description:

Parameters

context :

event :

`async` `Aggregator.`**`generate`**`(`*context*, *aggregated_event*, *depth*`)`

Description:

Parameters

context :

aggregated_event :

depth :

Bytes

String to Bytes Parser

`class` **`StringToBytesParser`**`(`*app*, *pipeline*, *id=None*, *config=None*`)`

Bases: *Processor*

Description:

** Default Config **

encoding : utf-8

`StringToBytesParser.`**`__init__`**`()`

Description:

String To Bytes Parser Method

StringToBytesParser.**process**(*context*, *event*)

Description:

Returns
event.decode(self.Encoding)

Bytes To String Parser

class BytesToStringParser(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description:

BytesToStringParser.**__init__**()

Description:

Bytes To String Parser Method

BytesToStringParser.**process**(*context*, *event*)

Description:

Returns
event.decode(self.Encoding)

Flatten

Flatten Dict Processor

class FlattenDictProcessor(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description: ....

Inspired by https://github.com/amirziai/flatten

Example:

“person”: {

“details”: {
“first_name”: “John”, “last_name”: “Doe”

}, “address”: {

“country”: “GB”, “city”: “London”, “postal_code”: “WC2N 5DU”

}

}

Gets converted to:

{
“person.details.first_name”: “John”, “person.details.last_name”: “Doe”, “person.address.country”: “GB”, “person.address.city”: “London”, “person.address.postal_code”: “WC2N 5DU”

}

..automethod:: bspump.common.FlattenDictProcessor.__init__()

Flatten Dict Processor

`FlattenDictProcessor.`**`flatten`**`(`*nested_dict*`)`

Description:

Returns
flattened_dict

`FlattenDictProcessor.`**`process`**`(`*context*, *event*`)`

Description:

Returns
event

Hexlify

Hexlify Processor

`class HexlifyProcessor``(`*app*, *pipeline*, *id=None*, *config=None*`)`

Bases: *Processor*

Description:

Hexlify Processor Method

HexlifyProcessor.**process**(*context*, *event*)

Description:

Returns
binascii.hexlify(event)

Iterator

Hexlify Processor

class IteratorSource(*app*, *pipeline*, *iterator: Iterator*, *id=None*, *config=None*)

Bases: *TriggerSource*

Description:

IteratorSource.**__init__**()

Description:

Hexlify Processor Method

async IteratorSource.**cycle**(**args*, ***kwags*)

Description:

Hexlify Processor

class IteratorGenerator(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Generator*

Description:

IteratorGenerator.__init__()

Description:

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

id
[str, default = None] ID

config
[JSON, defualt = None] configuration file containing additional information.

Iterator Generator Method

async IteratorGenerator.generate(*context*, *event*, *depth*)

Description:

Json

CySimd Json Parser

class CySimdJsonParser(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Fast JSON parser. Expects json bytes represented as bytes as input Based on https://github.com/TeskaLabs/cysimdjson

CySimdJsonParser.__init__()

Description: .

CySimd Json Parser Method

CySimdJsonParser.**process**(*context*, *event: bytes*)

Description:

Returns

self._parser.parse(event)

Std Dict To Json Parser

class StdDictToJsonParser(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description:

Std Dict To Json Parser Method

StdDictToJsonParser.**process**(*context*, *event*)

Description:

Returns

?

Std Json To Dict Parser

class StdJsonToDictParser(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description:

Std Json To Dict Parser Method

StdJsonToDictParser.**process**(*context*, *event*)

Description:

Returns

???

Dict To JsonBytes Parser

class DictToJsonBytesParser(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

DictToJsonBytesParser transforms a dictionary to JSON-string encoded in bytes. The encoding charset can be specified in the configuration in *encoding* field.

DictToJsonBytesParser.**__init__**()

Initializes the Parameters

Parameters

app
[object] Application object.

pipeline
[*Pipeline*] Name of the *Pipeline*.

id
[str, default=None,] ID of the class of config.

config
[JSON, or other compatible formats, default=None] Configuration file.

Dict To Json Bytes Parser Method

DictToJsonBytesParser.**process**(*context*, *event*)

Can be implemented to return event based on a given logic.

Parameters

context :
Additional information passed to the method.

event
[Data with time stamp stored in any data type, usually it is in JSON.] You can specify an event that is passed to the method.

Mapping

Mapping Keys Processor

class MappingKeysProcessor(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description: Mapping Keys Processor

Mapping Keys Processor Method

MappingKeysProcessor.**process**(*context*, *event: Mapping*) → list

Description: process is a method of a Mapping Keys Processor

Returns
event.keys()

Mapping Values Processor

class MappingValuesProcessor(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description:

Mapping Values Processor Method

MappingValuesProcessor.**process**(*context*, *event: Mapping*) → list

Description:

Returns
event.values()

Mapping Items Processor

class MappingItemsProcessor(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description:

Mapping Items Processor Method

MappingItemsProcessor.**process**(*context*, *event: Mapping*) → list

Description:

Returns
event.items()

Mapping Keys Generator

class MappingKeysGenerator(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Generator*

Description:

Mapping Keys Generator Method

async MappingKeysGenerator.**generate**(*context*, *event*, *depth*)

Description:

Null

Null Sink

class NullSink(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Sink*

Description:

Null Sink Method

NullSink.**process**(*context*, *event*)

Description:

Print

Print Sink

class PrintSink(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Bases: *Sink*

Description:

__init__(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Description:

Print Sink Method

PrintSink.**process**(*context*, *event*)

Description:

PPrint Sink

class PPrintSink(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Bases: *Sink*

Description:

__init__(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Description:

PPrint Sink Method

PPrintSink.**process**(*context*, *event*)

Description:

Print Processor

class PrintProcessor(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Bases: *Processor*

Description:

__init__(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Description:

Print Processor Method

PrintProcessor.**process**(*context*, *event*)

Description:

Returns

event

PPrint Processor

class PPrintProcessor(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Bases: *Processor*

Description:

__init__(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Description:

PPrint Processor Method

PPrintProcessor.**process**(*context*, *event*)

Description:

Returns
event

Print Context Processor

class PrintContextProcessor(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Bases: *Processor*

Description:

__init__(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Description:

Print Context Processor Method

PrintContextProcessor.**process**(*context*, *event*)

Description:

Returns
event

PPrint Context Processor

class PPrintContextProcessor(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Bases: *Processor*

Description:

__init__(*app*, *pipeline*, *id=None*, *config=None*, *stream=None*)

Description:

PPrint Context Processor Method

PPrintContextProcessor.**process**(*context*, *event*)

Description:

Returns
event

Routing

Direct Source

class DirectSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Source*

Description: This source processes inserted event synchronously.

DirectSource.__init__()

Description:

Direct Source

DirectSource.**put**(*context*, *event*, *copy_context=False*, *copy_event=False*)

This method serves to put events into the pipeline and process them right away.

Context can be an empty dictionary if is not provided.

async DirectSource.**main**()

Description:

Internal Source

class InternalSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Source*

Description:

InternalSource.__init__()

Description:

Internal Source methods

InternalSource.**put**(*context*, *event*, *copy_context=False*, *copy_event=False*)

Description: Context can be an empty dictionary if is not provided.

If you are getting a *asyncio.queues.QueueFull* exception, you likely did not implemented backpressure handling. The simpliest approach is to use RouterSink / RouterProcessor.

async InternalSource.**put_async**(*context*, *event*, *copy_context=False*, *copy_event=False*)

Description: This method allows to put an event into InternalSource asynchronously. Since a processing in the pipeline is synchronous, this method is useful mainly for situation, when an event is created outside of the pipeline processing. It is designed to handle situation when the queue is becoming full.

Context can be an empty dictionary if is not provided.

async InternalSource.**main**()

Description:

InternalSource.**rest_get**()

Description:

Returns
rest

Router Mix In

class RouterMixIn

Bases: object

Description: Router Mix in a class

RouterMixIn.**__init__**()

Router Mix In methods

RouterMixIn.**locate**(*source_id*)

Description:

Returns
source

RouterMixIn.**unlocate**(*source_id*)

Description: Undo locate() call, it means that it removes the source from a cache + remove throttling binds

Returns

??

RouterMixIn.**dispatch**(*context*, *event*, *source_id*, *copy_event=True*)

Description:

Returns
self.route(context, event, source_id, copy_event=True)

RouterMixIn.**route**(*context*, *event*, *source_id*, *copy_event=True*)

Description: This method routes an event to a InternalSource *source_id*.

It can be called multiple times from a process() method, which results in a cloning of the event.

Router Sink

class RouterSink(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Sink*, *RouterMixIn*

Description: Abstract Sink that dispatches events to other internal sources. One should override the process() method and call route() with target source id.

RouterSink.__init__()

Initializes the Parameters

Parameters

app
[object] Application object.

pipeline
[*Pipeline*] Name of the *Pipeline*.

id
[str, default=None,] ID of the class of config.

config
[JSON, or other compatible formats, default=None] Configuration file.

Router Processor

class RouterProcessor(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*, *RouterMixIn*

Description: Abstract Processor that dispatches events to other internal sources. One should override the process() method and call route() with target source id.

RouterProcessor.__init__()

Description:

Tee

Tee Source Processor

class TeeSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *InternalSource*

Description:

class SamplePipeline(bspump.Pipeline):

def __init__(self, app, pipeline_id):
super().__init__(app, pipeline_id)

self.build(
bspump.socket.TCPStreamSource(app, self, config={'port': 7000}),
bspump.common.TeeProcessor(app, self).bind("SampleTeePipeline.*TeeSource"),
bspump.common.PPrintSink(app, self)

)

class SampleTeePipeline(bspump.Pipeline):

```
    def __init__(self, app, pipeline_id):
        super().__init__(app, pipeline_id)

        self.build(
            bspump.common.TeeSource(app, self), bspump.common.PPrintSink(app, self)

        )
```

TeeSource.__init__()

Description:

Tee Source Method

TeeSource.bind(*target*)

Description:

Returns

async TeeSource.main()

Description:

Returns

Tee Processor

class TeeProcessor(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *RouterProcessor*

Description: See TeeSource for details.

TeeProcessor.__init__()

Description:

Tee Processor Method

TeeProcessor.**bind**(*target: str*)

Description: Target is a bspump.PumpService.locate() string

Returns

?

TeeProcessor.**unbind**(*target: str*)

Description:

Returns

?

TeeProcessor.**process**(*context*, *event*)

Description:

Returns

event

Time

Time Zone Normalizer

class TimeZoneNormalizer(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description: Normalizes datetime from local timezone (e.g. in config) to UTC, which is preferred internal datetime form

TimeZoneNormalizer.**__init__**()

Description:

Time Zone Normalizer Method

TimeZoneNormalizer.**normalize**(*local_time: datetime*) → datetime

Description: If *local_time* doesn't contain a time zone (e.g. it is naive), the timezone will be added from config

Returns
Normalized *local_time* in UTC

TimeZoneNormalizer.**process**(*context*, *event*)

Description: Abstract method to process the event. Must be customized.

Example:

```
>>> native_time = event["@timestamp"]
>>> local_time = self.normalize(native_time)
>>> event["@timestamp"] = local_time
```

Transfr

Mapping Transformator

class MappingTransformator(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Processor*

Description:

MappingTransformator.**__init__**()

Description:

Mapping Transformator Methods

MappingTransformator.**build**(*app*)

Description:

MappingTransformator.**process**(*context*, *event: Mapping*) → dict

Description:

Returns

dict(map(self._map, event.items()))

2.3.6 Advanced

BitSwan Pump provides more advanced Processors that can be used in a pipeline

Generator

Generator object is used to generate one or multiple events in asynchronous way

and pass them to following processors in the pipeline. In the case of Generator, user overrides *generate* method, not *process*.

1.) Generator can iterate through an event to create (generate) derived ones and pass them to following processors.

Example of a custom Generator class with generate method:

```
        class MyGenerator(bspump.Generator):

                async def generate(self, context, event, depth):
                        for item in event.items():
                                self.Pipeline.inject(context, item, depth)

2.) Generator can in the same way also generate completely independent events, if␣
↪necessary.
In this way, the generator processes originally synchronous events "out-of-band" e.g.␣
↪out of the synchronous processing within the pipeline.

Specific implementation of the generator should implement the generate method to process␣
↪events while performing
long running (asynchronous) tasks such as HTTP requests or SQL select.
The long running tasks may enrich events with relevant information, such as output of␣
↪external calculations.

Example of generate method:
```

```
async def generate(self, context, event, depth):

        # Perform possibly long-running asynchronous operation
        async with aiohttp.ClientSession() as session:
                async with session.get("https://example.com/resolve_color/{}".
↪format(event.get("color_id", "unknown"))) as resp:
                        if resp.status != 200:
                                return
                        new_event = await resp.json()

        # Inject a new event into a next depth of the pipeline
        self.Pipeline.inject(context, new_event, depth)
```

class Generator(*app*, *pipeline*, *id=None*, *config=None*)

Bases: ProcessorBase

Description:

Generator.**__init__**()

Description:

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

id
[str, default = None] ID

config
[JSON, defualt = None] configuration file containing additional information.

Generator Construction

Generator.**set_depth**(*depth*)

Description:

Parameters

depth : int

Generator.**process**(*context*, *event*)

Description:

Parameters

context :

event
[any data type] information of any data type with timestamp.

async Generator.**generate**(*context*, *event*, *depth*)

Description:

Parameters

context :

event
[any data type] information of any data type with timestamp.

depth : int

Analyzer

This is general analyzer interface, which can be the basement of different analyzers.
analyze_on_clock enables analyzis by timer, which period can be set by *analyze_period* or *Config["analyze_period"]*.

In general, the *Analyzer* contains some object, where it accumulates some information about events. Events go through analyzer unchanged, the information is recorded by *evaluate()* function. The internal object sometimes should be processed and sent somewhere (e.g. another pipeline), this process can be done by *analyze()* function, which can be triggered by time, pubsub or externally

class **Analyzer**(*app*, *pipeline*, *analyze_on_clock=False*, *id=None*, *config=None*)

Bases: *Processor*

Description:

Analyzer.**__init__**()

Initializes the Parameters

Parameters

app
[object] Application object.

pipeline
[*Pipeline*] Name of the *Pipeline*.

id
[str, default=None,] ID of the class of config.

config
[JSON, or other compatible formats, default=None] Configuration file.

Analyzer Construction

Analyzer.**start_timer**(*event_type*)

Description:

Analyzer

The main function, which runs through the analyzed object. Specific for each analyzer. If the analyzed object is *Matrix*, it is not recommended to iterate through the matrix row by row (or cell by cell). Instead use numpy fuctions. Examples: 1. You have a vector with n rows. You need only those row indeces, where the cell content is more than 10. Use *np.where(vector > 10)*. 2. You have a matrix with n rows and m columns. You need to find out which rows fully consist of zeros. use *np.where(np.all(matrix == 0, axis=1))* to get those row indexes. Instead *np.all()* you can use *np.any()* to get all row indexes, where there is at least one zero. 3. Use *np.mean(matrix, axis=1)* to get means for all rows. 4. Usefull numpy functions: *np.unique()*, *np.sum()*, *np.argmin()*, *np.argmax()*.

Analyzer.**analyze**()

Description:

Analyzer.**evaluate**(*context*, *event*)

The function which records the information from the event into the analyzed object.
Specific for each analyzer.

Parameters

context :

event
[any data type] information with timestamp.

Analyzer.**predicate**(*context*, *event*)

This function is meant to check, if the event is worth to process. If it is, should return True. specific for each analyzer, but default one always returns True.

Parameters

context :

event
[any data type] information with timestamp.

Returns
True

Analyzer.**process**(*context*, *event*)

The event passes through *process(context, event)* unchanged.
Meanwhile it is evaluated.

Parameters

context :

event
[any data type] information with timestamp.

Returns
event

async Analyzer.**on_clock_tick**()

Run analyzis every tick.

Analyzing Source

Lookup

Lookups serve for fast data searching in lists of key-value type. They can subsequently be localized and used in pipeline objects (processors and the like). Each lookup requires a statically or dynamically created value list.

If the "lazy" parameter in the constructor is set to True, no load method is called and the user is expected to call it when necessary.

class Lookup(*app*, *id=None*, *config=None*, *lazy=False*)

Bases: `Configurable`

Description:

Returns

Lookup.**__init__**()

Description:

Lookup Construction

Lookup.**time**()

Description:

Returns

time

Lookup.**ensure_future_update**(*loop*)

Description:

Returns

async Lookup.**load**() → bool

Description:

Lookup.**serialize**()

Description:

Lookup.**deserialize**(*data*)

Description:

Lookup.**is_master**()

Description:

Returns

??

MappingLookup

class MappingLookup(*app*, *id=None*, *config=None*, *lazy=False*)

Bases: Lookup, Mapping

Description:

MappingLookup.**__init__**()

Description:

Async Lookup Mixin

AsyncLookupMixin makes sure the value from the lookup is obtained asynchronously. AsyncLookupMixin is to be used for every technology that is external to BSPump, respective that require a connection to resource server such as SQL etc.

class AsyncLookupMixin(*app*, *id=None*, *config=None*, *lazy=False*)

Bases: Lookup

Description:

Dictionary Lookup

class DictionaryLookup(*app*, *id=None*, *config=None*, *lazy=False*)

Bases: MappingLookup

Description:

DictionaryLookup.**__init__**()

Description:

Dictionary Lookup Methods

DictionaryLookup.**__getitem__**(*key*)

DictionaryLookup.**__len__**()

DictionaryLookup.**serialize**()

Description:

Returns
json data

DictionaryLookup.**deserialize**(*data*)

Description:

DictionaryLookup.**rest_get**()

Description:

Returns
rest

DictionaryLookup.**set**(*dictionary: dict*)

Description:

Lookup Provider

class LookupProviderABC(*lookup*, *url*, *id=None*, *config=None*)

Bases: ABC, Configurable

Description:

LookupProviderABC.**__init__**()

Description:

Lookup Provider Methods

async LookupProviderABC.**load**()

Description:

Lookup BatchProvider ABC

class LookupBatchProviderABC(*lookup*, *url*, *id=None*, *config=None*)

Bases: LookupProviderABC, ABC

Description:

LookupBatchProviderABC.**__init__**()

Description:

Anomaly

class Anomaly

Bases: dict

Description: Anomaly is an abstract class to be overriden for a specific anomaly and its type.

Returns

Implement: TYPE, on_tick

Anomaly.**__init__**()

2.3.7 Technologies

Technologies Reference Documentation describes the *Technologies* section.

Apache Kafka

Connection

class KafkaConnection(*app*, *id=None*, *config=None*)

Bases: *Connection*

KafkaConnection serves to connect BSPump application with an instance of Apache Kafka messaging system. It can later be used by processors to consume or provide user-defined messages.

```
config = {"compression_type": "gzip"}
app = bspump.BSPumpApplication()
svc = app.get_service("bspump.PumpService")
svc.add_connection(
        bspump.kafka.KafkaConnection(app, "KafkaConnection", config)
)
```

`ConfigDefaults` options:

compression_type (str): Kafka supports several compression types: `gzip`, `snappy` and `lz4`.
This option needs to be specified in Kafka Producer only, Consumer will decompress automatically.

security_protocol (str): Protocol used to communicate with brokers.
Valid values are: PLAINTEXT, SSL. Default: PLAINTEXT.

sasl_mechanism (str): Authentication mechanism when security_protocol
is configured for SASL_PLAINTEXT or SASL_SSL. Valid values are: PLAIN, GSSAPI, SCRAM-SHA-256, SCRAM-SHA-512. Default: PLAIN

sasl_plain_username (str): username for sasl PLAIN authentication.
Default: None

sasl_plain_password (str): password for sasl PLAIN authentication.
Default: None

`KafkaConnection.__init__()`

initializes variables

Parameters

app
[Application] Name of the Application.

id
[, default = None] ID information.

config
[JSON or txt, default= None] Configuration file of any supported type.

connection Methods

`async KafkaConnection.create_producer(**kwargs)`

Creates a Producer.

Parameters

****kwargs :**
Additional information can be passed to this method.

Returns
producer

`KafkaConnection.create_consumer(*topics, **kwargs)`

Creates a consumer.

Parameters

***topics :**
any number of topics can be passed to this method.

**kwargs :
additional information can be passed to this method.

Returns
consumer

KafkaConnection.**get_bootstrap_servers**()
Returns parsed bootstrap servers found in config.

Returns
list of url

KafkaConnection.**get_compression**()
Returns compression type to use in connection

Returns
compression_type

Source

class KafkaSource(*app*, *pipeline*, *connection*, *id=None*, *config=None*)
Bases: *Source*

KafkaSource object consumes messages from an Apache Kafka system, which is configured in the KafkaConnection object. It then passes them to other processors in the pipeline.

```
class KafkaPipeline(bspump.Pipeline):

        def __init__(self, app, pipeline_id):
                super().__init__(app, pipeline_id)
                self.build(
                        bspump.kafka.KafkaSource(app, self, "KafkaConnection",␣
↪config={'topic': 'messages'}),
                        bspump.kafka.KafkaSink(app, self, "KafkaConnection", config=
↪{'topic': 'messages2'}),
                )

To ensure that after restart, pump will continue receiving messages where it left␣
↪of, group_id has to
be provided in the configuration.

When the group_id is set, the consumer group is created and the Kafka server will␣
↪then operate
```

(continues on next page)

(continued from previous page)

```
in the producer-consumer mode. It means that every consumer with the same group_id␣
↪will be assigned
unique set of partitions, hence all messages will be divided among them and thus␣
↪unique.

Long-running synchronous operations should be avoided or places inside the␣
↪OOBGenerator in the asynchronous
way or on thread using ASAB Proactor service (see bspump-oob-proactor.py example in
↪"examples" folder).
Otherwise, the session_timeout_ms should be raised to prevent Kafka from␣
↪disconnecting the consumer
from the partition, thus causing rebalance.
```

KafkaSource.**__init__**()

Initializes parameters.

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

connection
[Connection] information needed to create a connection.

id : , default = None

config : , default = None

Source Methods

KafkaSource.**create_consumer**()

Creates a consumer.

async KafkaSource.**initialize_consumer**()

Creates a consumer after the loop is started.

async KafkaSource.**main**()

Method that starts the Source.

Sink

class KafkaSink(*app*, *pipeline*, *connection*, *key_serializer=None*, *id=None*, *config=None*)

Bases: *Sink*

Description: KafkaSink is a sink processor that forwards the event to a Apache Kafka specified by a KafkaConnection object.

KafkaSink expects bytes as an input. If the input is string or dictionary, it is automatically transformed to bytes using encoding charset specified in the configuration.

```
class KafkaPipeline(bspump.Pipeline):

	def __init__(self, app, pipeline_id):
		super().__init__(app, pipeline_id)
		self.build(
			bspump.kafka.KafkaSource(app, self, "KafkaConnection",␣
↪config={'topic': 'messages'}),
			bspump.kafka.KafkaSink(app, self, "KafkaConnection", config=
↪{'topic': 'messages2'}),
		)

There are two ways to use KafkaSink:
- Specify a single topic in KafkaSink config - topic, to be used for all the events␣
↪in pipeline.
- Specify topic separetly for each event in event context - context['kafka_topic'].
	Topic from configuration is than used as a default topic.
	To provide business logic for event distribution, you can create topic␣
↪selector processor.
Processor example:
```

```
class KafkaTopicSelector(bspump.Processor):

	def process(self, context, event):
		if event.get("weight") > 10:
			context["kafka_topic"] = "heavy"
		else:
			context["kafka_topic"] = "light"

		return event

Every kafka message can be a key:value pair. Key is read from event context -␣
↪context['kafka_key'].
If kafka_key is not provided, key defaults to None.
```

KafkaSink.**__init__**()

Initilizes the parameters that are passed to the Sink class.

Parameters

app
: [Application] Name of the Application.

pipeline
: [Pipeline] Name of the Pipeline.

connection
: [Connection] information needed to create a connection.

key_serializer : , default = None

id : , default = None

config : , default = None

Sink Methods

KafkaSink.**process**(*context*, *event: Union[dict, str, bytes]*)

Outputs events to a chosen location.

Parameters

context
[type] Additional information.

event:typing.Union[dict, str, bytes] :

Key Filter Kafka

class KafkaKeyFilter(*app*, *pipeline*, *keys*, *id=None*, *config=None*)

Bases: *Processor*

KafkaKeyFilter reduces the incoming event stream from Kafka based on a key provided in each event.

Every Kafka message has a key, KafkaKeyFilter selects only those events where the event key matches one of provided 'keys', other events will be discarded.

Set filtering *keys* as a parameter (in bytes) in the KafkaKeyFilter constructor.

KafkaKeyFilter is meant to be inserted after KafkaSource in a Pipeline.

KafkaKeyFilter.**__init__**()

Initializes variables

Parameters

app
[Application] Name of the **`Application <https://asab.readthedocs.io/en/latest/asab/application.html`_**.

pipeline
[Pipeline] Name of the Pipeline.

keys
[bytes] keys used to filter out events from the event stream.

id : , default = None

config
[JSON, default = None] configuration file in JSON

KafkaKeyFilter.**process**(*context*, *event*)

Does the filtering processed based on passed key variable.

Parameters

context
[Context] additional information passed to the method

event : any type,a single unit of information that flows through the Pipeline.

Batch Sink

class KafkaBatchSink(*app*, *pipeline*, *connection*, *key_serializer=None*, *id=None*, *config=None*)

Bases: KafkaSink

KafkaBatchSink is a sink processor that forwards the event to an Apache Kafka specified by a KafkaConnection object in batches.

It is a proof of concept sink, that allows faster processing of events in the pipeline, but does not guarantee processing of all events in situations when the pump is closed etc.

There is a work to be done with cooperation with aiokafka, so the send_and_wait method works properly and is able to send events in batches.

KafkaBatchSink.**__init__**()

Initializing parameters passed to the BatchSink class.

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

connection
[Connection] Information needed to creates connection.

key_serializer : ,default None

id : , default = None

config
[JSON, default = None] Configuration file with additional information.

Batch Sink Methods

KafkaBatchSink.**process**(*context*, *event: Union[dict, str, bytes]*)

Starts the sink process.

Parameters

context
[type?] Additional information.

event: typing.Union[dict, str, bytes] : type?

Topic Initializer

class KafkaTopicInitializer(*app*, *connection*, *id: Optional[str] = None*, *config: Optional[dict] = None*)

Bases: Configurable

KafkaTopicInitializer reads topic configs from file or from Kafka sink/source configs, checks if they exists and creates them if they don't.

KafkaAdminClient requires blocking connection, which is why this class doesn't use the connection module from BSPump.

Usage: topic_initializer = KafkaTopicInitializer(app, "KafkaConnection") topic_initializer.include_topics(MyPipeline) topic_initializer.initialize_topics()

KafkaTopicInitializer.**__init__**()

Initializes the parameters passed to the class.

Parameters

app
[Application] Name of the Application.

connection
[Connection] Information needed to create a connection.

id: typing.Optional[str] = None :

config: dict = None
[JSON] configuration file containing important information.

topic initializer methods

KafkaTopicInitializer.**include_topics**(*, *topic_config=None*, *kafka_component=None*, *pipeline=None*, *config_file=None*)

Includes topic from config file or dict object. It can also scan Pipeline and get topics from Source or Sink.

Parameters

- :

topic_config
[, default= None] Topic config file.

kafka_component : , default= None

pipeline
[, default= None] Name of the Pipeline.

config_file
[, default= None] Configuration file.

KafkaTopicInitializer.**include_topics_from_file**(*topics_file: str*)

Includes topics from a topic file.

Parameters

topics_file:str
[str] Name of a topic file we wanted to include.

KafkaTopicInitializer.**include_topics_from_config**(*config_object*)

Includes topics using a config

Parameters

config_object
[JSON] config object containing information about what topics we want to include.

KafkaTopicInitializer.**fetch_existing_topics**()

???

KafkaTopicInitializer.**check_and_initialize**()

Initializes new topics and logs a warning.

KafkaTopicInitializer.**initialize_topics**()

Initializes topics ??

Elastic Search

Elastic Search is a Analytics and full-text search engine. Commonly used for Application Performance Management mainly Analysis of Logs.

Source

ElasticSearchSource is using standard Elastic's search API to fetch data.

configs

index - Elastic's index (default is 'index-*').

scroll_timeout - Timeout of single scroll request (default is '1m'). Allowed time units: https://www.elastic.co/guide/en/elasticsearch/reference/current/common-options.html#time-units

specific pamameters

paging - boolean (default is True)

request_body - dictionary described by Elastic's doc: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-body.html

Default is:

```
default_request_body = {
        'query': {
                'bool': {
                        'must': {
                                'match_all': {}
                        }
                }
        },
}
```

class ElasticSearchSource(*app*, *pipeline*, *connection*, *request_body=None*, *paging=True*, *id=None*, *config=None*)

Bases: *TriggerSource*

Description:

ElasticSearchSource.**__init__**()

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

connection
[Connection] Information of the connection.

request_body JSON, default = None
Request body needed for the request API call.

paging : ?, default = True

id
[ID, default = None] ID

config
[JSON/dict, default = None] Configuration file with additional information.

Source Methods

`async ElasticSearchSource.cycle()`
Gets data from Elastic and injects them into the pipeline.

ElasticSearch Aggs Source

ElasticSearchAggsSource is used for Elastic's search aggregations.

configs

index: - Elastic's index (default is 'index-*').

specific pamameters

request_body dictionary described by Elastic's doc: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-body.html

Default is:

```
default_request_body = {
        'query': {
                'bool': {
                        'must': {
                                'match_all': {}
                        }
                }
        },
}
```

`class ElasticSearchAggsSource(app, pipeline, connection, request_body=None, id=None, config=None)`
Bases: *TriggerSource*

Description:

`ElasticSearchAggsSource.__init__()`
Description:

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

connection
[Connection] Information of the connection.

request_body JSON, default = None
Request body needed for the request API call.

id
[ID, default = None] ID info

config
[JSON/dict, default = None] configuration file with additional information.

ElasticSearch Aggs Source Methods

`async ElasticSearchAggsSource.cycle()`
Sets request body and path to create query call.

`async ElasticSearchAggsSource.process_aggs(path, aggs_name, aggs)`
Description:

Parameters

path :

aggs_name :

agss :

`async ElasticSearchAggsSource.process_buckets(path, parent, buckets)`
Recursive function for buckets processing. It iterates through keys of the dictionary, looking for 'buckets' or 'value'. If there are 'buckets', calls itself, if there is 'value', calls process_aggs and sends an event to process

Parameters

path :

parent :

buckets :

ElasticSearch Connection

ElasticSearchConnection allows your ES source, sink or lookup to connect to ElasticSearch instance

usage:

```
# adding connection to PumpService
svc = app.get_service("bspump.PumpService")
svc.add_connection(
        bspump.elasticsearch.ElasticSearchConnection(app, "ESConnection")
)
```

```
# pass connection name ("ESConnection" in our example) to relevant BSPump's object:

self.build(
```

(continues on next page)

(continued from previous page)

```
                bspump.kafka.KafkaSource(app, self, "KafkaConnection"),
                bspump.elasticsearch.ElasticSearchSink(app, self, "ESConnection")
)
```

class ElasticSearchConnection(*app*, *id=None*, *config=None*)

Bases: *Connection*

Description:

Sample Config

url
['''http'://{ip/localhost}:{port}'] URL of the source. Could be multi-URL. Each URL should be separated by ';' to a node in ElasticSearch cluster.

username
['string' , default = ' '] Used when authentication is required

password
['string', default = ' '] Used when authentication is required

loader_per_url
[int, default = 4] Number of parallel loaders per URL.

output_queue_max_size
[int, default = 10] Maximum queue size.

bulk_out_max_size
[? * ? * ?, default = 12 * 1024 * 1024] ??

timeout
[int, default = 300] Timout value.

fail_log_max_size
[int, default = 20] Maximum size of failed log messages.

precise_error_handling
[bool, default = False] If True all Errors will be logged, If false soft errors will be omitted in the Logs.

ElasticSearchConnection.**__init__**()

Description:

Parameters

app
[Application] Name of the Application

id
[ID, default= None] ID

config
[JSON or dict, default= None] configuration file with additional information for the methods.

ElasticSearch Connection Methods

ElasticSearchConnection.**get_url**()

Returns
list of URLS of nodes connected to the cluster

ElasticSearchConnection.**get_session**()

Returns current Client Session Authentication and Loop

Returns
aiohttp.ClientSession(auth=self._auth, loop=self.Loop)

ElasticSearchConnection.**consume**(*index*, *data_feeder_generator*, *bulk_class=<class 'bspump.elasticsearch.connection.ElasticSearchBulk'>*)

Checks the content of data_feeder_generator and bulk and if There is data to be send it calls enqueue method.

Parameters

index :

data_feeder_generator :

bulk_class=ElasticSearchBulk :
creates a instance of the ElasticSearchBulk class

ElasticSearchConnection.**flush**(*forced=False*)

It goes through the list of bulks and calls enqueue for each of them.

Parameters

forced : bool, default = False

ElasticSearchConnection.**enqueue**(*bulk*)

Properly enqueue the bulk.

Parameters

bulk :

Elastic Search Bulk

class ElasticSearchBulk(*connection*, *index*, *max_size*)

Bases: object

Description:

ElasticSearchBulk.**__init__**()

Initializes the variables

Parameters

connection
[Connection] Name of the Connection.

index
[str] ???

max_size
[int] Maximal size of bulks.

Elastic Search Bulk Methods

ElasticSearchBulk.**consume**(*data_feeder_generator*)

Appends all items in data_feeder_generator to Items list. Consumer also resets Aging and Capacity.

Parameters

data_feeder_generator
[list] list of our data that will be passed to a generator and later Uploaded to ElasticSearch.

Returns
self.Capacity <= 0

async ElasticSearchBulk.**upload**(*url*, *session*, *timeout*)

Uploads data to Elastic Search.

Parameters

url
[string] Uses URL from config to connect to ElasticSearch Rest API.

session
[?] ?

timeout
[int] uses timout value from config. Value of time for how long we want to be connected to ElasticSearch.

Returns

?

ElasticSearchBulk.**partial_error_callback**(*response_items*)

Description: When an upload to ElasticSearch fails for error items (document could not be inserted), this callback is called.

Parameters

response_items :

Parameters
response_items – list with dict items: {"index": {"_id": ..., "error": ...}}

ElasticSearchBulk.**full_error_callback**(*bulk_items*, *return_code*)

Description: When an upload to ElasticSearch fails b/c of ElasticSearch error, this callback is called.

Parameters

bulk_items
[list] list with tuple items: (_id, data)

return_code :
ElasticSearch return code

Returns
False if the bulk is to be resumbitted again

Lookup

class ElasticSearchLookup(*app*, *connection*, *id=None*, *config=None*, *cache=None*, *lazy=False*)

Bases: MappingLookup, AsyncLookupMixin

The lookup that is linked with a ES. It provides a mapping (dictionary-like) interface to pipelines. It feeds lookup data from ES using a query. It also has a simple cache to reduce a number of database hits.

configs

index - Elastic's index

key - field name to match

scroll_timeout - Timeout of single scroll request (default is '1m'). Allowed time units: https://www.elastic.co/guide/en/elasticsearch/reference/current/common-options.html#time-units

Example:

```
The ElasticSearchLookup can be then located and used inside a custom enricher:

        class AsyncEnricher(bspump.Generator):

                def __init__(self, app, pipeline, id=None, config=None):
                        super().__init__(app, pipeline, id, config)
                        svc = app.get_service("bspump.PumpService")
                        self.Lookup = svc.locate_lookup("MySQLLookup")

                async def generate(self, context, event, depth):
                        if 'user' not in event:
                                return None

                        info = await self.Lookup.get(event['user'])

                        # Inject a new event into a next depth of the pipeline
                        self.Pipeline.inject(context, event, depth)
```

ElasticSearchLookup.**__init__**()

Description:

Parameters

app
[Application] Name of the Application.

connection
[Connection] Name of the Connection

id
[ID, default= None] ID

config
[JSON, default= None] Configuration file with additional information.

cache : ?,default= None

lazy : ?, default= None

Lookup methods

async ElasticSearchLookup.**get**(*key*)

Obtain the value from lookup asynchronously.

Parameters

key : ?

Returns
value

ElasticSearchLookup.**build_find_one_query**(*key*) → dict

Override this method to build your own lookup query

Parameters

key : ?

Returns
Default single-key query

async ElasticSearchLookup.**load**()

Sets the length of Cache to Count.

Returns
True

classmethod ElasticSearchLookup.**construct**(*app*, *definition: dict*)

Constructs config, id, and connection based on config.

Parameters

app
[Application] Name of the Application.

definition:dict
[Definition] Definition containing information about certain variables.

Returns
cls(app, newid, connection, config)

Sink

class ElasticSearchSink(*app*, *pipeline*, *connection*, *id=None*, *config=None*, *bulk_class=<class 'bspump.elasticsearch.connection.ElasticSearchBulk'>*, *data_feeder=<function data_feeder_create_or_index>*)

Bases: *Sink*

ElasticSearchSink allows you to insert events into ElasticSearch through POST requests

The following attributes can be passed to the context and thus override the default behavior of the sink:

es_index (STRING): ElasticSearch index name

data_feeder accepts the event as its only parameter and yields data as Python generator The example implementation is:

def data_feeder_create_or_index(event):
_id = event.pop("_id", None)

if _id is None:
yield b'{"create":{}}

'

else:

yield orjson.dumps(
{"index": {"_id": _id}}, option=orjson.OPT_APPEND_NEWLINE

)

yield orjson.dumps(event, option=orjson.OPT_APPEND_NEWLINE)

ElasticSearchSink.__init__()

Description:

Parameters

app
[Application] Name of the Application

pipeline
[Pipeline] Name of the Pipeline

connection
[Connection] Name of the Connection

id
[ID, default= None] ID

config
[JSON, default= None] Configuration file with additional information.

bulk_class=ElasticBulk :

data_feeder=data_feeder_create_or_index :

Sink methods

ElasticSearchSink.**process**(*context*, *event*)

Description:

Parameters

context :

event
[any data type] Information with timestamp.

Data Feeder Methods

data_feeder.**data_feeder_create_or_index**()

Creates an index.

Parameters

event
[Data with time stamp stored in any data type usually is in JSON.] You can specify an event that is passed to the method.

data_feeder.**data_feeder_create**()

Creates a data feeder.

Parameters

event
[Data with time stamp stored in any data type usually is in JSON.] You can specify an event that is passed to the method.

data_feeder.**data_feeder_index**()

Description:

Parameters

event
[Data with time stamp stored in any data type usually is in JSON.] You can specify an event that is passed to the method.

data_feeder.**data_feeder_update**()

Updates data feeder.

Parameters

event
[Data with time stamp stored in any data type usually is in JSON.] You can specify an event that is passed to the method.

data_feeder.**data_feeder_delete**()

Deletes data feeder.

Parameters

event
[Data with time stamp stored in any data type usually is in JSON.] You can specify an event that is passed to the method.

Files

File ABC Source

class FileABCSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *TriggerSource*

Description:

FileABCSource.**__init__**()

Description:

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

id
[ID, default = None] ID

config
[JSON, default = None] Configuration file with additional information.

File ABC Source Methods

async FileABCSource.**cycle**()

Cycles through a file.

async FileABCSource.**simulate_event**()

The simulate_event method should be called in read method after a file line has been processed.

It ensures that all other asynchronous events receive enough time to perform their tasks. Otherwise, the application loop is blocked by a file reader and no other activity makes a progress.

async FileABCSource.**read**(*filename*, *f*)

Description: Override this method to implement your File Source. *f* is an opened file object.

Parameters

filename
[file] Name of the file.

f :

File Block Source

class FileBlockSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *FileABCSource*

Description:

FileBlockSource.**__init__**()

Description:

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

id
[ID, default = None] ID

config
[JSON, default = None] Configuration file with additional information.

async FileBlockSource.**read**(*filename*, *f*)

Loads a file.

Parameters

filename
[file] Name of the file.

f :

File Block Sink

class FileBlockSink(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Sink*

Description:

** Config Defaults **

path : ''

mode : wb

flags : O_CREAT

FileBlockSink.**__init__**()

Parameters

app
[Application] Name of the Application

pipeline
[Pipeline] Name of the Pipeline.

id
[ID, default = None] ID

config
[JSON, default = None] Configuration file with additional information.

`FileBlockSink.get_file_name(context, event)`

Override this method to gain control over output file name.

Parameters

context :

event
[any type] a single unit of information that is propagated through the pipeline

Returns
config path

`FileBlockSink.process(context, event)`

Opens a file.

Parameters

context :

event
[any type] a single unit of information that is propagated through the pipeline

File csv Source

`class FileCSVSource(app, pipeline, fieldnames=None, id=None, config=None)`

Bases: *FileABCSource*

Description:

`FileCSVSource.__init__()`

Description:

Parameters

app
[Application] Name of the Application.

pipeline
[Pipeline] Name of the Pipeline.

id
[ID, default = None] ID

config
[JSON, default = None] Configuration file with additional information.

`FileCSVSource.reader(f)`

Description:

Parameters

f :

Returns

??

async FileCSVSource.**read**(*filename*, *f*)

Description:

Parameters

filename :

f :

File csv Sink

class **FileCSVSink**(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Sink*

Description:

** Default Config**

path : ‘’

dialect : ‘excel’

delimiter : ‘,’

doublequote : True

escapechar : “”

lineterminator : os.linesep

quotechar : “”’

quoting : csv.QUOTE_MINIMAL

skipinitialspace : False

strict : False

FileCSVSink.**__init__**()

Description:

FileCSVSink.**get_file_name**(*context*, *event*)

Description: Override this method to gain control over output file name.

Parameters

context :

event :

Returns

path of context and config

FileCSVSink.**writer**(*f*, *fieldnames*)

Description:

Parameters

f :

fieldnames
[file] Name of the file.

Returns
dialect and fieldnames

FileCSVSink.**process**(*context*, *event*)

Description:

Parameters

context :

event
[any data type] Information with timestamp.

FileCSVSink.**rotate**()

Description: Call this to close the currently open file.

File json Source

class FileJSONSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *FileABCSource*

Description: This file source is optimized to load even large JSONs from a file and parse that. The loading & parsing is off-loaded to the worker thread so that it doesn't block the IO loop.

FileJSONSource.**__init__**()

Description:

Parameters

app :

pipeline :

id
[ID, default= None] ID

config
[JSON, default = None] configuration file with additional information

async FileJSONSource.**read**(*filename*, *f*)

Description:

Parameters

filename :

f :

File line Source

class FileLineSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *FileABCSource*

Description:

FileLineSource.**__init__**()

Description:

Parameters

app: Application
Name of the Application

pipeline
[Pipeline] Name of the Pipeline

id : ID, default = None

config
[JSON, default = None] Configuration file with additional information

async FileLineSource.**read**(*filename*, *f*)

Description:

Parameters

filename :

f :

File Multiline Source

class FileMultiLineSource(*app*, *pipeline*, *separator*, *id=None*, *config=None*)

Bases: *FileABCSource*

Description: Read file line by line but try to join multi-line events by separator. Separator is a (fixed) pattern that should present at the begin of the line, if it is a new event.

Example: <133>1 2018-03-24T02:37:01+00:00 machine program 22068 - Start of the multiline event

2nd line of the event 3rd line of the event

<133>1 2018-03-24T02:37:01+00:00 machine program 22068 - New event

The separatpr is '<' string in this case

FileMultiLineSource.__init__()

Description:

Parameters

app: Application
Name of the Application

pipeline
[Pipeline] Name of the Pipeline

separator :

id : ID, default = None

config
[JSON, default = None] Configuration file with additional information

async FileMultiLineSource.read(*filename*, *f*)

Description:

Parameters

filename :

f :

Lookup Provider

class FileBatchLookupProvider(*lookup*, *url*, *id=None*, *config=None*)

Bases: LookupBatchProviderABC

Loads lookup data from a file on local filesystem.

FileBatchLookupProvider.__init__()

Description:

async FileBatchLookupProvider.load()

Description:

Returns
result

FileBatchLookupProvider.load_on_thread()

Description:

InfluxDB

Connection

class InfluxDBConnection(*app*, *id=None*, *config=None*)

Bases: *Connection*

Description: InfluxDBConnection serves to connect BSPump application with an InfluxDB database. The InfluxDB server is accessed via URL, and the database is specified using the *db* parameter in the configuration.

```
app = bspump.BSPumpApplication()
svc = app.get_service("bspump.PumpService")
svc.add_connection(
        bspump.influxdb.InfluxDBConnection(app, "InfluxConnection1")
)

**Config Default**

url : http://localhost:8086/

db : mydb

output_queue_max_size : 10

output_bucket_max_size : 1000 * 1000

timout : 30

retry_enabled : False

response_codes_to_retry : 404, 502, 503, 504
```

InfluxDBConnection.**__init__**()

Description:

Parameters

app
[Application] Name of the Application.

id : ID, default = None

config
[JSON, default = None] Configuration file with additional information.

Sink

class InfluxDBSink(*app*, *pipeline*, *connection*, *id=None*, *config=None*)

Bases: *Sink*

Description: InfluxDBSink is a sink processor, that stores the event into an InfluxDB database specified in the InfluxDBConnection object.

```
class SamplePipeline(bspump.Pipeline):

        def __init__(self, app, pipeline_id):
                super().__init__(app, pipeline_id)
                self.build(
                        bspump.socket.TCPStreamSource(app, self, config={'port':␣
↪7000}),
                        bspump.influxdb.InfluxDBSink(app, self, "InfluxConnection1")
                )
```

InfluxDBSink.**__init__**()

Description:

Parameters

app :

pipeline :

connection :

id : ID, default = None

config : str,JSON, default = None

InfluxDBSink.**process**(*context*, *event*)

Description:

Parameters

context :

event
[any data type] Information with timestamp.

IPC and Socket

Datagram

class DatagramSource(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Source*

Description:

DatagramSource.**__init__**()

Description:

async DatagramSource.**main**()

Description:

Datagram sink

class DatagramSink(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Sink*

Description:

DatagramSink.**__init__**()

Description:

DatagramSink.**process**(*context*, *event*)

Description:

Protocol

class SourceProtocolABC(*app*, *pipeline*, *config*)

Bases: object

Source protocol is a handler class, that basically gets the socket (in reader) and extract the payload from it in a way that is conformant to expected protocol.

That is happening in the *handle()* method. The output is to be shipped to source.process() method.

SourceProtocolABC.**__init__**()

Description:

async SourceProtocolABC.**handle**(*source*, *stream*, *context*)

Description:

Line Source Protocol

class LineSourceProtocol(*app*, *pipeline*, *config*)

Bases: *SourceProtocolABC*

Description: Basically readline() for reading lines from a socket.

LineSourceProtocol.**__init__**()

Description:

async LineSourceProtocol.**handle**(*source*, *stream*, *context*)

Description:

Stream

class Stream(*loop*, *socket*, *outbound_queue=None*)

Bases: object

Description: This object represent a client connection. It is unencrypted STREAM socket.

Stream.**__init__**()

async Stream.**recv_into**(*buf*)

Stream.**send**(*data*)

async Stream.**outbound**()

Handle outbound direction

async Stream.**close**()

TLS Stream

class TLSStream(*loop*, *sslcontext*, *socket*, *server_side: bool*)

Bases: object

Description: This object represent a TLS client connection. It is encrypted SSL/TLS socket abstraction.

TLSStream.**__init__**()

Description:

async TLSStream.**recv_into**(*buf*)

Description:

TLSStream.**send**(*data*)

Description:

async TLSStream.**outbound**()

Handle outbound direction

async TLSStream.**close**()

Description:

Steam Server Source

class StreamServerSource(*app*, *pipeline*, *id=None*, *config=None*, *protocol_class=<class 'bspump.ipc.protocol.LineSourceProtocol'>*)

Bases: *Source*

Description:

StreamServerSource.**__init__**()

Description:

StreamServerSource.**start**(*loop*)

Description:

async StreamServerSource.**stop**()

Description:

async StreamServerSource.**main**()

Description:

Stream Client Sink

class StreamClientSink(*app*, *pipeline*, *id=None*, *config=None*)

Bases: *Sink*

Description:

StreamClientSink.**__init__**()

Description:

StreamClientSink.**process**(*context*, *event*)

Description:

FTP

connection

source

RabbitMQ / AMQP

Source

class AMQPSource(*app*, *pipeline*, *connection*, *id=None*, *config=None*)

Bases: *Source*

Description:

AMQPSource.**__init__**()

Set the initial ID, *Pipeline* and Task.

Parameters

app
[Application] Name of an *Application* <https://asab.readthedocs.io/en/latest/asab/application.html#>`_ .

pipeline
[address of a pipeline] Name of a *Pipeline*.

id
[str, default None] Name of a the *Pipeline*.

config
[compatible config type , default None] Option for adding a configuration file.

async AMQPSource.**main**()

Description:

async AMQPSource.**process_message**(*method*, *properties*, *body*)

Description:

classmethod AMQPSource.**construct**(*app*, *pipeline*, *definition: dict*)

Description:

AMQP Full Message Source

class AMQPFullMessageSource(*app*, *pipeline*, *connection*, *id=None*, *config=None*)

Bases: *AMQPSource*

Description:

AMQPFullMessageSource.**process_message**(*method*, *properties*, *body*)

Description:

Sink

class AMQPSink(*app*, *pipeline*, *connection*, *id=None*, *config=None*)

Bases: *Sink*

AMQPSink.**__init__**()

Initializes the Parameters

Parameters

app
[object] Application object.

pipeline
[*Pipeline*] Name of the *Pipeline*.

id
[str, default=None,] ID of the class of config.

config
[JSON, or other compatible formats, default=None] Configuration file.

AMQPSink.**process**(*context*, *event*)

Can be implemented to return event based on a given logic.

Parameters

context :
Additional information passed to the method.

event
[Data with time stamp stored in any data type, usually it is in JSON.] You can specify an event that is passed to the method.

Connection

class AMQPConnection(*app*, *id=None*, *config=None*)

Bases: *Connection*

AMQPConnection.**__init__**()

Description:

Parameters

app
[Application] Specification of an Application.

id : default None

config
: [JSON or other compatible format, default None] It contains important information and data responsible for creating a connection.

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W