databay

Release 0.3.0

Voy Zan

Jun 21, 2021

CONTENTS

Databay is a Python interface for **scheduled data transfer**.

It facilitates transfer of (any) data from A to B, on a scheduled interval.

CHAPTER

ONE

GITHUB PAGE

```
pip install databay
```

- *Overview* - Learn what is Databay.
- *Examples* - See Databay in use.
- *Extending Databay* - Use Databay in your project.
- *API Reference* - Read the API documentation.

CHAPTER

TWO

FEATURES

Simple, decoupled interface	Easily implement *data production* and *consumption* that fits your needs.
Granular control over data transfer	Multiple ways of *passing information* between producers and consumers.
Asyncio supported	You can *produce* or *consume* asynchronously.
We'll handle the rest	*Scheduling*, *startup and shutdown*, *exception handling*, *logging*.
Support for custom scheduling	Use *your own scheduling logic* if you like.

A simple example:

```
# Data producer
inlet = HttpInlet('https://some.test.url.com/')

# Data consumer
outlet = MongoOutlet('databay', 'test_collection')

# Data transfer between the two
link = Link(inlet, outlet, datetime.timedelta(seconds=5))

# Start scheduling
planner = ApsPlanner(link)
planner.start()
```

Every 5 seconds this snippet will pull data from a test URL, and write it to MongoDB.

Explore this documentation:

2.1 Key Concepts

2.1.1 Overview

Databay is a Python interface for scheduled data transfer.

It facilitates transfer of (any) data from A to B, on a scheduled interval.

In Databay, data transfer is expressed with three components:

- `Inlets` - for data production.
- `Outlets` - for data consumption.
- `Links` - for handling the data transit between inlets and outlets.

Scheduling is implemented using third party libraries, exposed through the `BasePlanner` interface. Currently two BasePlanner implementations are available - using Advanced Python Scheduler (`ApsPlanner`) and Schedule (`SchedulePlanner`).

A simple example:

```
# Create an inlet, outlet and a link.
http_inlet = HttpInlet('https://some.test.url.com/')
mongo_outlet = MongoOutlet('databay', 'test_collection')
link = Link(http_inlet, mongo_outlet, datetime.timedelta(seconds=5))

# Create a planner, add the link and start scheduling.
planner = ApsPlanner(link)
planner.start()
```

Every 5 seconds this snippet will pull data from a test URL, and write it to MongoDB.

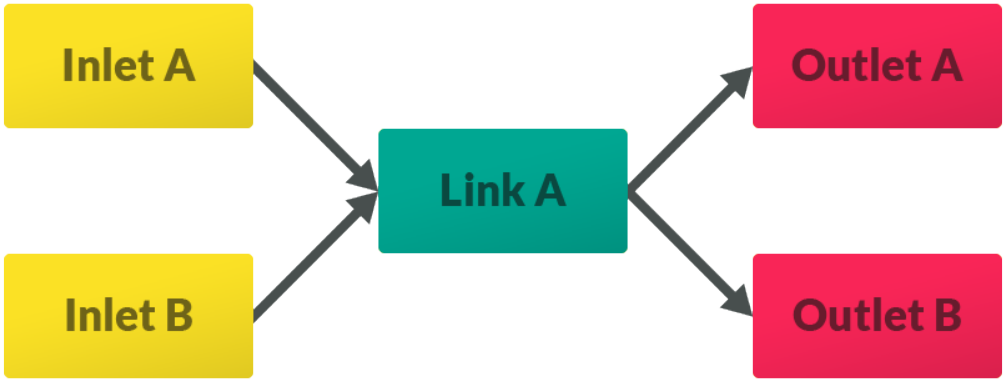
While Databay comes with a handful of built-in inlets and outlets, its strength lies in extendability. To use Databay in your project, create concrete implementations of `Inlet` and `Outlet` classes that handle the data production and consumption functionality you require. Databay will then make sure data can repeatedly flow between the inlets and outlets you create. *Extending Inlets* and *extending Outlets* is easy and has a wide range of customization. Head over to *Extending Databay* section for a detailed explanation.

2.1.2 Inlets, Outlets and Links

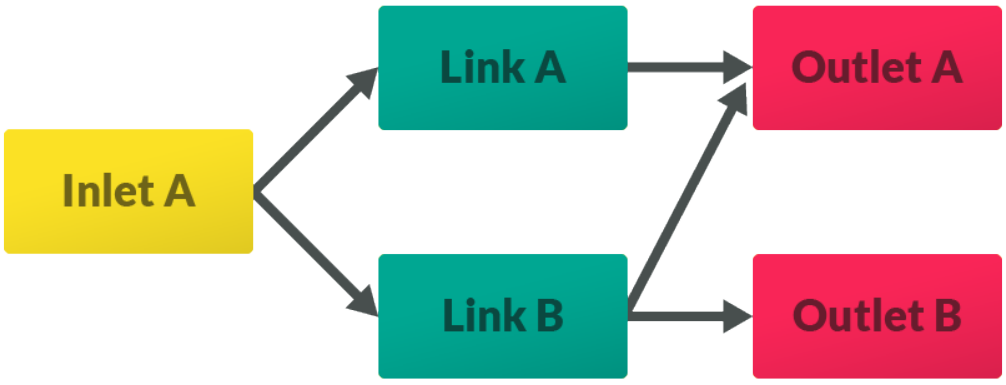
Databay treats data transfer as a unidirectional graph, where data flows from `Inlet` nodes to `Outlet` nodes. An example of an inlet and outlet could be an HTTP request client and a CSV writer respectively.

The relationship between the inlets and outlets is explicitly defined as a `Link`.

One link may connect multiple inlets and outlets.



One inlet or outlet can be connected through multiple links.



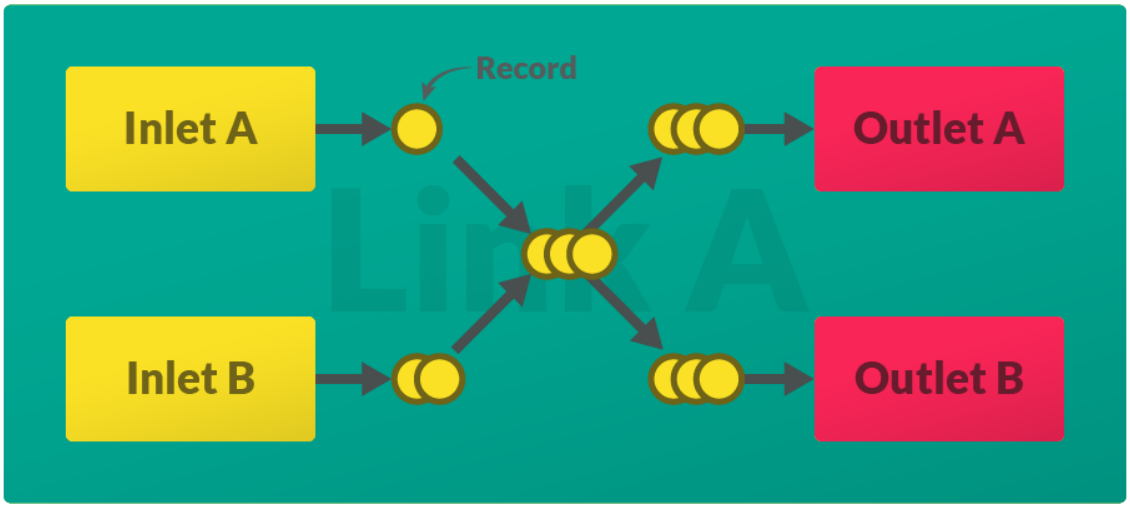
2.1.3 Link transfer

One cycle of data production, propagation and consumption is called *transfer*. During transfer, a link will pull data from all its inlets and then push that collected data to all its outlets.

Each link contains an interval at which it will run the data transfer. This interval is specified on construction with the `interval` parameter of type `datetime.timedelta`.

```
Link([inlets], [outlets], interval=datetime.timedelta(minutes=10))
```

One quantity of data handled by Databay is represented with a *`Record`*.



Both pulling and pushing is executed asynchronously, yet pushing only starts once all inlets have finished returning their data.

There's a lot more you can do to your data during a transfer - such as filtering, buffering, grouping and transforming. Head over to *Advanced Concepts* to learn more.

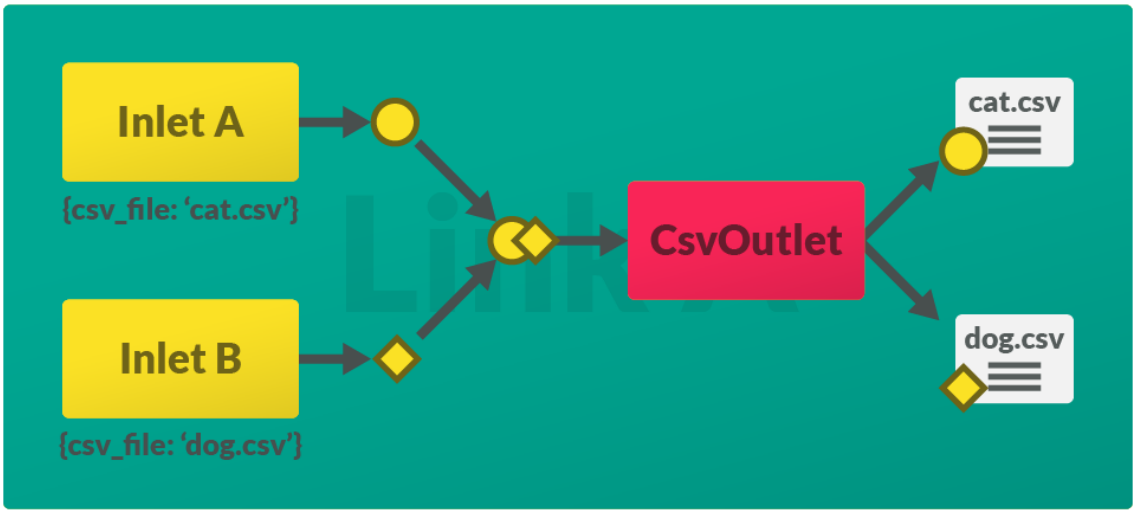
Transfer Update

Each transfer is identified by a unique *`Update`* object that is available to all inlets and outlets affected by that transfer. It contains the tags of the governing link (if specified) and an incremental integer index. Use the `str(update)` to get a formatted string of that update.

```
# for link called 'twitter_link' and the 16th transfer execution.
>>> print(update)
twitter_link.16
```

2.1.4 Records

`Records` are data objects that provide a unified interface for data handling across Databay. In addition to storing data produced by inlets, records may also carry individual metadata. This way information can be passed between inlets and outlets, facilitating a broad spectrum of custom implementations. For instance one CsvOutlet could be used for writing into two different csv files depending on which inlet the data came from.



2.1.5 Scheduling

The principal functionality of Databay is to execute data transfer repeatedly on a pre-defined interval. To facilitate this, links are governed by a scheduler object implementing the *BasePlanner* class. Using the concrete scheduling functionality, links' transfers are executed in respect with their individual interval setting.

To schedule a link, all you need to do is to add it to a planner and call *start* to begin scheduling.

```
link = Link(some_inlet, some_outlet, timedelta(minutes=1))
planner = SchedulePlanner(link)
planner.start()
```

Databay provides two built-in *BasePlanner* implementations based on two popular Python scheduling libraries:

- *ApsPlanner* - using Advanced Python Scheduler.
- *SchedulePlanner* - using Schedule.

While they differ in the method of scheduling, threading and exception handling, they both cover a reasonable variety of scheduling scenarios. Please refer to their appropriate documentation for more details on the difference between the two.

You can easily use a different scheduling library of your choice by extending the *BasePlanner* class and implementing the link scheduling and unscheduling yourself. See *Extending BasePlanner* for more.

2.1.6 Start and shutdown

Start

To begin scheduling links you need to call *start* on the planner you're using. Both *ApsPlanner* and *SchedulePlanner* handle *start* as a synchronous blocking function. To run *start* without blocking the current thread, wrap its call within a new thread or a process:

```
th = Thread(target=planner.start)
th.start()
```

Shutdown

To stop scheduling links you need to call *shutdown(wait:bool=True)* on the planner you're using. Note that this may or may not let the currently transferring links finish, depending on the implementation of the *BasePlanner* that you're using. Both *ApsPlanner* and *SchedulePlanner* allow waiting for the links if *shutdown* is called passing `True` as the `wait` parameter.

on_start and on_shutdown

Just before scheduling starts, *Inlet.on_start* and *Outlet.on_start* callbacks will be propagated through all inlets and outlets. Consequently, just after scheduling shuts down, *Inlet.on_shutdown* and *Outlet.on_shutdown* callbacks will be propagated through all inlets and outlets. In both cases, these callbacks will be called only once for each inlet and outlet. Override these callback methods to implement custom starting and shutdown behaviour in your inlets and outlets.

immediate_transfer

By default BasePlanner will execute *Link.transfer* function on all its links once upon calling *BasePlanner.start*. This is to avoid having to wait for the link's interval to expire before the first transfer. You can disable this behaviour by passing `immediate_transfer=False` parameter on construction.

2.1.7 Exception handling

If exceptions are thrown during transfer, both planners can be set to log and ignore these by passing the `ignore_exceptions=True` parameter on construction. This ensures transfer of remaining links can carry on even if some links are erroneous. If exceptions aren't ignored, both *ApsPlanner* and *SchedulePlanner* will log the exception and gracefully shutdown.

Additionally, each *Link* can be configured to catch exceptions by passing `ignore_exceptions=True` on construction. This way any exceptions raised by individual inlets and outlets can be logged and ignored, allowing the remaining nodes to continue execution and for the transfer to complete.

```
# for planners
planner = SchedulePlanner(ignore_exceptions=True)
planner = ApsPlanner(ignore_exceptions=True)

# for links
link = Link(..., ignore_exceptions=True)
```

2.1.8 Logging

All classes in Databay are configured to utilise a `Python Logger` called `databay` or its child loggers. Databay utilises a custom `StreamHandler` with the following signature:

```
%Y-%m-%d %H:%M:%S.milis|levelname| message (logger name)
```

For example:

```
2020-07-30 19:51:41.318|D| http_to_mongo.0 transfer (databay.Link)
```

By default Databay will only log messages with `WARNING` priority or higher. You can manually enable more verbose logging by calling:

```
logging.getLogger('databay').setLevel(logging.DEBUG)

# Or do it only for a particular child logger:

logging.getLogger('databay.ApsPlanner').setLevel(logging.DEBUG)
```

You can attach new handlers to any of these loggers in order to implement custom logging behaviour - such as a `FileHandler` to log into a file, or a separate `StreamHandler` to customise the print signature.

2.1.9 Motivation

The data flow in Databay is different from a more widely adopted Observer Pattern, where data production and propagation are represented by one object, and consumption by another. In Databay data production and propagation is split between the *`Inlet`* and *`Link`* objects. This results in a data flow model in which each stage - data transfer, production and consumption - is independent from the others. *`Inlets`* are only concerned with producing data, *`Outlets`* only with consuming data and *`Links`* only with transferring data. Such a model is motivated by separation of concerns and by facilitating custom implementation of data producers and consumers.

Next Steps

1. Learn about extending *Inlets* and *Outlets*.
2. See the *Examples*

2.2 Advanced Concepts

Explore advanced concepts of Databay. These sections assume you're already familiar with the *Key Concepts*.

2.2.1 Processors

Processors are a middleware pipeline that alters the records transferred from inlets to outlets. Two most common usages of these would be:

- Filtering - removing some or all records before feeding them to outlets.
- Transforming - altering the records before feeding them to outlets.

Simple example

```
# Example filtering
def only_large_numbers(records: List[Records]):
    result = []
    for record in records:
        if record.payload >= 100:
            result.push(record)
    return result

# Example transforming:
def round_to_integers(records: List[Records]):
    for record in records:
        record.payload = round(record.payload)
    return records

# pass to a link
link = Link(..., processors=[only_large_numbers, round_to_integers])
```

The processor pipeline used in the above example will turn the following list:

```
[99.999, 200, 333.333]
```

into:

```
[200, 333]
```

Note that `99.999` got filtered out given the order of processors. If we were to swap the processors, the rounding would occur before filtering, allowing all three results through the filter.

Processors explained

A processor is a `callable` function that accepts a list of records and returns a (potentially altered) list of records.

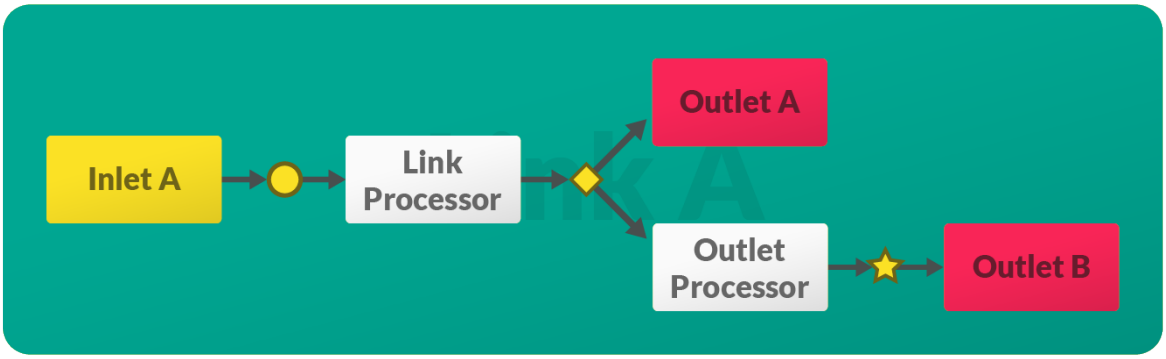
Processors are called in the order in which they are passed, after all inlets finish producing their data. The result of each processor is given to the next one, until finally the resulting records continue the transfer normally.

Link vs Outlet processors

Databay supports two types of processors, depending on the scope at which they operate.

- *`Link processor`* - applied to all records transferred by that link.
- *`Outlet processor`* - applied only to records consumed by the particular outlet.

This distinction can be used to determine at which level a particular processor is to be applied.



Observe in the diagram above that the *Outlet A* will receive records modified only by the Link Processor. At the same time, the *Outlet B* will receive records modified first by the Link Processor, then by the Outlet Processor.

For example:

- *Link processor* - A filtering processor that removes duplicate records produced by an inlet could be applied to all records at link level.

```
def remove_duplicates(records: List[Record]):
    result = []
    for record in records:
        if record not in result:
            result.append(record)
    return result

link = Link(..., processors=remove_duplicates)
```

- *Outlet processor* - A filtering processor that filters out records already existing in a CSV file could be applied only to the CsvOutlet, preventing duplicate records from being written to a CSV file, yet otherwise allowing all records to be consumed by the other outlets in the link.

```
def filter_existing(records: List[Record]):
    with open(os.fspath('./data/records.csv'), 'r') as f:
        reader = csv.DictReader(csv_file)
        existing = []
        for row in reader:
            for key, value in row.items():
                existing.append(value)

    result = []
    for record in records:
        if record.payload not in existing:
            result.append(record)
    return result

csv_outlet = CsvOutlet(..., processors=filter_existing)
link = Link(inlets, csv_outlet, ...)
```

Link processors are used before *Groupers*, while Outlet processors are used after.

Best practices

Responsibility

Databay doesn't make any further assumptions about processors - you can implement any type of processors that may suit your needs. This also means Databay will not ensure the records aren't corrupted by the processors, therefore you need to be conscious of what each processor does to the data.

If you wish to verify the integrity of your records after processing, attach an additional processor at the end of your processor pipeline that will validate the correctness of your processed records before sending it off to the outlets.

2.2.2 Groupers

By default outlets will be given all produced records at the same time. Groupers are a middleware which allows you to break that list of records into batches. Each batch is then fed into outlets separately, allowing outlets to process an entire batch individually at the same time, instead of processing each record one by one.

Simple example

Following grouper will group the records into batches based on their payload 'name' attribute.

```
def grouper_by_name(batches: List[List[Record]]):
    result = []
    for batch in batches:
        new_batch = {}
        for record in batch:
            new_batch[record.payload['name']] = record

        result.append(list(new_batch.values()))

link = Link(..., groupers=grouper_by_name)
```

Which will turn the following list of records:

```
[
    {'name': 'a', 'value': 1},
    {'name': 'a', 'value': 2},
    {'name': 'b', 'value': 3},
    {'name': 'b', 'value': 4}
]
```

into the following list of batches:

```
[
    [
        {'name': 'a', 'value': 1},
```

(continues on next page)

(continued from previous page)

```
        {'name': 'a', 'value': 2}
    ],
    [
        {'name': 'b', 'value': 3},
        {'name': 'b', 'value': 4}
    ]
]
```

Groupers explained

A **grouper** is a `callable` function that accepts a list of batches and returns a list of batches with a different shape.

A **list of batches** is a two-dimensional list containing *`Records`* grouped into sub-lists.

Each of these sub-lists is called a **batch**.

Fox example:

1. Consider an inlet that produces six records with a simple payload. This first list is a **list of records**, as all records are contained within it.

   ```
   [0,1,2,3,4,5]
   ```

2. When grouped by a pairing grouper, that list may be turned into the following two-dimensional list. This second list is a **list of batches**, as it contains the records grouped into three sub-lists.

   ```
   [[0,1], [2,3], [3,4]]
   ```

3. Each element of the list of batches is a **batch**, as it represents one sub-list containing the records.

   ```
   [0,1]
   ```

Note that:

- All records contained in all batches should equal to the list of records.
- All groupers are called with a list of batches. Especially note that this includes the first grouper, which is provided with a list of batches containing one batch with all the records. This is due to the fact that groupers are order-agnostic, allowing you to swap them around expecting a consistent behaviour. Therefore all groupers should expect a list of batches and be aware that its shape may vary.

After batching

Once records are grouped into batches, each batch is fed into the outlets as if it was an individual list of records. Depending on the particular implementation, outlets may expect that and process the entire batch at the same time. If a particular outlet doesn't support batch processing, the result of batching will effectively be nullified except for the order in which the records will be consumed.

The following examples illustrate how the records are fed into the outlets with and without groupers.

Without groupers:

```
print(records)
# [0,1,2,3,4,5]

for outlet in self.outlets:
    outlet.push(records)
```

In this case `outlet.push` is called once with the entire list of records `[0,1,2,3,4,5]`.

With groupers:

```
print(records)
# [0,1,2,3,4,5]

batches = [records] # the default batch contains all records
for grouper in groupers:
    batches = grouper(batches) # the groupers process the batches

print(batches)
# [[0,1],[2,3],[4,5]]

for batch in batches:
    for outlet in self.outlets:
        outlet.push(batch)
```

In this case `outlet.push` is called three times, each time receiving a different batch: `[0,1]`, `[2,3]` and `[4,5]`.

Observe that when no groupers are provided, there is only one batch containing all records. This will provide all outlets with all records at the same time, effectively nullifying the batches' functionality described in this section.

Best practices

Responsibility

Databay doesn't make any assumptions about groupers - you can implement any type of groupers that may suit your needs. This also means Databay will not ensure the records aren't corrupted by the groupers. Therefore you need to be conscious of what each grouper does to the data.

Only batching

Note that you should only use groupers' functionality to group the records into batches. Do not transform or filter the records using groupers - you can use *Processors* for that instead. Hypothetically, if a list of batches produced by any grouper was to be flattened it should return the list of records as originally produced by the inlets, except for the order of records.

```
print(records)
# [0,1,2,3,4,5]

batches = [records] # the default batch contains all records
for grouper in groupers:
    batches = grouper(batches)

flat_batches = [record for batch in batches for record in batch] # flatten the batches

# do both list contain same elements regardless of the order?
print(set(records) == set(flat_batches))
# True
```

Adhere to correct structure

Databay expects to work with either one- or two-dimensional data, depending on whether groupers are used. One-dimensional being a list of records (ie. without batching), two-dimensional being a list of batches (ie. with batching). In either case, outlets will be provided with a list (or sub-list) of records and are expected to process these as a one-dimensional list.

Introducing further sub-list breakdowns - eg. batches containing batches - is not expected and such subsequent subdivisions will not be indefinitely iterated. If you choose to introduce further subdivisions ensure the outlets you use are familiar with such data structure and are able to process it accordingly.

2.2.3 Buffers

`Buffers` are special built-in *Processors*. They allow you to temporarily accumulate records before passing them over to outlets.

Simple example

The following example uses a buffer to store the records until the number of records produced exceed 10 items.

1. Define a buffer with `count_threshold=10`:

```
buffer = Buffer(count_threshold=10)
link = Link(inlet, outlet, processors=buffer)
```

1. On first transfer the inlet produces 4 records, the buffer stores them. The outlet receives no records.
2. On second transfer the inlet produces 4 records, the buffer stores them along with the first 4. The outlet still receives no records.
3. On third transfer the inlet produces 3 records. Having exceeded the count_threshold of `10`, the buffer will release all 11 records to the outlet. The outlet receives a list of 11 records.

Store or release?

When processing records (see *Processors*) a *Buffer* will figure out whether records should be stored or released. This is done by passing the list of records to *Buffer's* internal `callable` functions called controllers.

Each controller performs different types of checks, returning `True` or `False` depending on whether records should be released or stored respectively.

Default controllers

Buffer comes with two default controllers:

- `count_controller` - buffering records until reaching a count threshold defined by `Buffer.count_threshold` parameter, counted from the first time the records are stored. For example:

```
buffer = Buffer(count_threshold=50) # release records every 50 records.
```

- `time_controller` - buffering records until reaching a time threshold defined by `Buffer.time_threshold` parameter, counted from the first time the records are stored. For example:

```
buffer = Buffer(time=60) # release records every 60 seconds.
```

Custom controllers

Apart from using the default controllers, buffer accepts any number of custom controllers. Each controller will be called with the list of records and is expected to return `True` or `False` depending on whether records should be buffered or released. For example:

```
def big_value_controller(records: List[Records]):
    for record in records:
        if record.payload.value > 10000
            return True

    return False

buffer = Buffer(custom_controllers=big_value_controller)
```

Buffer reset

Every time the records are released, the buffer will reset the counters of its default controllers and empty the list of records stored.

You can pass a `callable` as an optional *on_reset* parameter, which will be invoked every time *Buffer.reset* is called.

Combining controllers

You can use any combination of default and custom controllers. *Buffer* allows you to use two types of boolean logic when evaluating whether records should be released:

- conjunction (AND) - releasing records only when **all** controllers return True.
- disjunction (OR) - releasing records as soon as **any** controller returns True (default).

For example:

```
buffer = Buffer(count_threshold=10, time_threshold=60, conjugate_
↪controllers=False) # OR
```

This buffer will release records once 10 records were produced or 60 seconds have elapsed - whichever comes first.

```
buffer = Buffer(count_threshold=10, time_threshold=60, conjugate_
↪controllers=True) # AND
```

This buffer will release records once 10 records were produced and 60 seconds have elapsed.

The order of execution of controllers is as follows:

1. Custom controllers, in order they are passed to the *Buffer*.
2. Count controller.
3. Time controller.

Buffer uses short-circuit logic to stop evaluation of controllers as soon as the decision to release or store is known, therefore not all controllers may be called each time the *Buffer* is executed.

Once the records are released, the *buffer will reset*.

Flush

Buffer contains a boolean field called `flush`, which if set to `True` will enforce release of records, independently of what the controllers may decide. Such flushing will only take place next time the buffer is called during the upcoming transfer. Flushing will also *reset the buffer*.

Best practices

One-to-one relationship

Given the internal record storage functionality, one buffer should only be used as either *a Link or an Outlet processor* - but never both at the same time.

Similarly, one buffer should only be used on one *Link* or *Outlet* - never multiple at the same time.

Ensure records are consumed

Note that in several scenarios a buffer may never release its records, therefore they would never be consumed by the outlets. Consider the following examples:

- Databay crashes before records are released.
- Planner is stopped before records are released.
- Thresholds are set to unreachable numbers

Databay does not automatically handle such occasions, however you may preempt these and ensure that records are released manually by combining the buffer's `flush` functionality with planners' *`force_transfer`* method.

```
try:
    # set up Databay
    buffer = Buffer(count_threshold=4000)
    link = Link(inlets, outlets, interval=10)
    planner = SchedulePlanner(link)
    planner.start()

except Exception as e:
    print('Error while running Databay: ' + str(e))

finally:
    buffer.flush = True # ensure the buffer will release data
    link.remove_inlets(link.inlets) # we don't need to produce any more data
    planner.force_transfer() # run one final transfer to flush the data
```

2.3 Extending Databay

In order to handle your custom data production and consumption in Databay, you need to extend the *`Inlet`* and *`Outlet`* classes.

2.3.1 Extending Inlets

To implement custom data production you need to extend the *`Inlet`* class, override the *`Inlet.pull`* method and return the data produced.

Simple example

1. Extend the *Inlet* class, returning produced data from the *pull* method:

```
class RandomIntInlet(Inlet):

    def pull(self, update):
        return random.randint(0, 100)
```

1. Instantiate it:

```
random_int_inlet = RandomIntInlet()
```

1. Add it to a link and start scheduling:

```
link = Link(random_int_inlet,
            print_outlet,
            interval=timedelta(seconds=5),
            tags='random_ints')

planner = SchedulePlanner(link)
planner.start()
```

Above setup will produce a random integer every 5 seconds (*See full example*).

Each pull call is provided with an *Update* object as a parameter. It contains the tags of the governing link (if specified) and an incremental integer index. Use the `str(update)` to get a formatted string of that update. See *Transfer Update* for more.

Your inlet may skip producing data by returning an empty `list`.

Creating records

Data produced by inlets is wrapped in *Record* objects before being passed to outlets. If you wish to control how records are created or attach local metadata, use the *Inlet.new_record* method to create records within your inlet and return these instead.

```
class RandomIntInlet(Inlet):

    def pull(self, update):
        new_integer = random.randint(0, 100)
        record = self.new_record(payload=new_integer)
        return record
```

Producing multiple records

During one transfer you may produce multiple data entities within the *Inlet.pull* method. Returning a `list` is an indication that multiple records are being produced at once, in which case each element of the `list` will be turned into a *Record*. Any return type other than `list` (eg. `tuple`, `set`, `dict`) will be considered as one *Record*.

Returning a `list`, producing two records:

```
def pull(self, update):

    # produces two records
    return [random.randint(0, 50), random.randint(0, 100)]
```

Returning a `set`, producing one record:

```
def pull(self, update):

    # produces one records
    return {random.randint(0, 50), random.randint(0, 100)}
```

Same is true when explicitly creating multiple records within *`pull`* and returning these.

```
def pull(self, update):
    first_record = self.new_record(random.randint(0, 50))
    second_record = self.new_record(random.randint(0, 100))

    return [first_record, second_record]
```

If you wish for one record to contain a `list` of data that doesn't get broken down to multiple records, you can either create the record yourself passing the `list` as payload or return a nested `list`:

```
def pull(self, update):
    r1 = random.randint(0, 50)
    r2 = random.randint(0, 100)

    return self.new_record(payload=[r1, r2])

# or
...

def pull(self, update):
    r1 = random.randint(0, 50)
    r2 = random.randint(0, 100)

    return [[r1, r2]]
```

Global metadata

`Inlet` can attach custom metadata to all records it produces. Metadata's intended use is to provide additional context to records when they are consumed by outlets. To do so, when constructing an *`Inlet`* pass a metadata dictionary, a copy of which will be attached to all records produced by that *`Inlet`* instance.

```
random_cat_inlet = RandomIntInlet(metadata={'animal': 'cat'})
# produces Record(metadata={'animal': 'cat'})

random_parrot_inlet = RandomIntInlet(metadata={'animal': 'parrot'})
# produces Record(metadata={'animal': 'parrot'})
```

Metadata dictionary is independent from the inlet that it is given to. Inlet should not modify the metadata or read it; instead inlets should expect all setup parameters to be provided as arguments on construction.

Incorrect:

```
def MyInlet():
    def __init__(self, metadata):
        self.should_do_stuff = metadata.get('should_do_stuff')
```

Correct:

```
def MyInlet():
    def __init__(self, should_do_stuff, *args, **kwargs):
        super().__init__(*args, **kwargs) # metadata dict gets passed and stored here
        self.should_do_stuff = should_do_stuff
```

Metadata supported by each outlet differs and is dependent on the particular outlet implementation. Please refer to specific outlet documentation for more information on metadata expected.

Additionally, each record is supplied with a special `__inlet__` metadata entry containing string representation of the inlet that produced it.

```
>>> record.metadata['__inlet__']
RandomIntInlet(metadata={})
```

Local metadata

Apart from providing an inlet with *Global metadata* that will be the same for all records, you may also attach local per-record metadata that can vary for each record. This can be done inside of the *pull* method by specifying a metadata dictionary when creating a record using *Inlet.new_record* method.

```
class RandomIntInlet(Inlet):

    def pull(self, update):
        new_integer = random.randint(0, 100)

        if new_integer > 50:
            animal = 'cat'
        else:
            animal = 'parrot'

        record = self.new_record(payload=new_integer, metadata={'animal': animal})
        return record
```

Note that local metadata will override global metadata if same metadata is specified globally and locally.

Start and shutdown

All inlets contain *Inlet.active* flag that is set by the governing link when scheduling starts and unset when scheduling stops. You can use this flag to refine the behaviour of your inlet.

You can further control the starting and shutting down functionality by overriding the *Inlet.on_start* and *Inlet.on_shutdown* methods. If one *Inlet* instance is governed by multiple links, these callbacks will be called only once per instance by whichever link executes first.

```
class RandomIntInlet(Inlet):

    def pull(self, update):
        return random.randint(0, 100)

    def on_start(self):
        random.seed(42)
```

Asynchronous inlet

You may implement asynchronous data production by defining *Inlet.pull* as a coroutine. The governing link will await all its inlets to finish producing their data before passing the results to outlets.

```
import asyncio
from databay import Inlet

class AsyncInlet(Inlet):

    # Note the 'async' keyword
    async def pull(self, update):
        async_results = await some_async_code()
        return async_results
```

See *Basic Asynchronous* for a full example of implementing asynchronous code in Databay.

You can limit (throttle) how many inlets can execute simultaneously by setting *inlet_concurrency* parameter when constructing a link.

Test your inlet

Databay comes with a template `unittest.TestCase` designed to validate your implementation of *Inlet* class. To use it, create a new test class extending *InletTester* and implement *InletTester.get_inlet* method returning an instance of your inlet.

```
from databay.misc import inlet_tester

class RandomIntInletTest(inlet_tester.InletTester):

    def get_inlet(self):
        return RandomIntInlet()

    ...

    # You can add further tests here
```

You may also return a `list` of inlets, to run each test on various configurations of your inlet:

```
def get_inlet(self):
    return [
        RandomIntInlet(),
        RandomIntInlet(min=10, max=200),
    ]
```

Running such a concrete test will execute a variety of test cases that ensure your inlet correctly provides the expected functionality. These include:

- Creating new records.
- Attaching global and local metadata.
- Calling *pull* method.

Since *InletTester* will call *pull* on your inlet, you may want to mock some of your inlet's functionality in order to separate testing of its logic from external code.

Next Steps

1. Learn about extending *Outlets*.
2. See the *Examples*

2.3.2 Extending Outlets

To implement custom data consumption you need to extend the `Outlet` class and override the `Outlet.push` method.

Simple example

1. Extend the `Outlet` class, printing the incoming data in the `push` method:

```
class PrintOutlet(Outlet):

    def push(self, records: [Record], update):
        for record in records:
            print(update, record.payload)
```

1. Instantiate it:

```
print_outlet = PrintOutlet()
```

1. Add it to a link and and schedule:

```
link = Link(random_int_inlet,
            print_outlet,
            interval=timedelta(seconds=2),
            tags='print_outlet')

planner = SchedulePlanner(link)
planner.start()
```

Above setup will print all records transferred by that link (*See full example*).

Each push call is provided with an `Update` object as one of parameters. It contains the tags of the governing link (if specified) and an incremental integer index. Use the `str(update)` to get a formatted string of that update. See *Transfer Update* for more.

Consuming Records

Outlets are provided with a `list` of all records produced by all inlets of the governing link. Each *`Record`* contains two fields:

1. *`Record.payload`* - data stored in the record.
2. *`Record.metadata`* - metadata attached to the record

```
from databay import Outlet

class ConditionalPrintOutlet(Outlet):

    def push(self, records, update):
        for record in records:
            if record.metadata.get('should_print', False):
                print(record.payload)
```

By default a copy of records is provided to outlets in order to prevent accidental data corruption. You can disable this mechanism by passing `copy_records=False` when constructing a link, in which case the same `list` will be provided to all outlets. Ensure you aren't modifying the records or their underlying data in your *`Outlet.push`* method.

Metadata

Your outlet can be built to behave differently depending on the metadata carried by the records. Metadata is attached to each record when inlets produce data. Learn more about the difference between *Global metadata* and *Local metadata*.

When creating an outlet it is up to you to ensure the expected metadata and its effects are clearly documented. To prevent name clashes between various outlets' metadata, it is recommended to include outlet name in the metadata keys expected by your outlet.

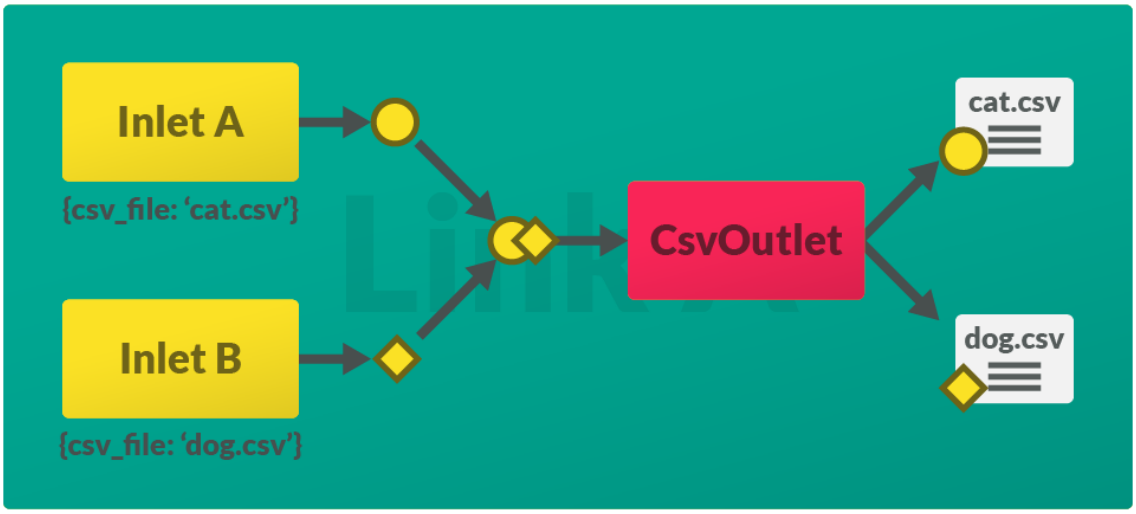
Incorrect:

```
CSV_FILE = 'CSV_FILE'
```

Correct:

```
CSV_FILE = 'CsvOutlet.CSV_FILE'
```

```
class CsvOutlet(Outlet):

    # Name of csv file to write records to.
    CSV_FILE = 'CsvOutlet.CSV_FILE'

    def push(self, records:[Record], update):
        for record in records:
            if self.CSV_FILE in record.metadata:
                csv_file = record.metadata[self.CSV_FILE] + '.csv'

                ...
                # write to csv_file specified

...

random_int_inletA = RandomIntInlet(metadata={CsvOutlet.CSV_FILE: 'cat'})
random_int_inletB = RandomIntInlet(metadata={CsvOutlet.CSV_FILE: 'dog'})
```



For clarity and readability, Databay provides the `MetadataKey` type for specifying metadata key class attributes.

```
from databay.outlet import MetadataKey

class CsvOutlet(Outlet):
    CSV_FILE:MetadataKey = 'CsvOutlet.CSV_FILE'
```

Start and shutdown

All outlets contain *`Outlet.active`* flag that is set by the governing link when scheduling starts and unset when scheduling stops. You can use this flag to refine the behaviour of your outlet.

You can further control the starting and shutting down functionality by overriding the *`Outlet.on_start`* and *`Outlet.on_shutdown`* methods. If one *`Outlet`* instance is governed by multiple links, these callbacks will be called only once per instance by whichever link executes first.

```
class PrintOutlet(Outlet):

    def push(self, records, update):
        print(f'{self.prefix} - {records}')

    def on_start(self):
        self.prefix = 'foo'
```

Asynchronous outlet

You may implement asynchronous data consumption by defining *`Outlet.push`* as a coroutine.

```
import asyncio
from databay import Outlet

class AsyncOutlet(Outlet):

    # Note the 'async' keyword
    async def push(self, records, update):
        async_results = await some_async_code(records)
        await asyncio.sleep(1)
```

See *Basic Asynchronous* for a full example of implementing asynchronous code in Databay.

Next Steps

1. Learn about extending *Inlets*.
2. See the *Examples*

2.3.3 Extending BasePlanner

Databay comes with two implementations of BasePlanner - *ApsPlanner* and *SchedulePlanner*. If you require custom scheduling functionality outside of these two interfaces, you can create your own implementation of *BasePlanner*. Have a look at the two existing implementations for reference: ApsPlanner and SchedulePlanner.

To extend the *BasePlanner* you need to provide a way of executing *Link.transfer* method repeatedly by implementing the following methods. Note that some of these methods are private since they are called internally by BasePlanner and should not be executed directly.

- _schedule
- _unschedule
- _start_planner
- _shutdown_planner
- running

_schedule

Schedule a *Link*. This method runs whenever *add_links* is called and should not be executed directly. It should accept a link and add the *Link.transfer* method to the scheduling system you're using. Note that you do not need to store the link in your planner - BasePlanner will automatically store it under *BasePlanner.links* when *add_links* is called. It isn't required for the scheduling to be already running when `_schedule` is called.

Each link comes with a `datetime.timedelta` interval providing the frequency at which its *Link.transfer* method should be run. Use *Link.interval* and schedule according to the interval specified.

If the scheduler you're using utilises some form of task-managing job objects, you must assign these to the link being scheduled using *Link.set_job*. This is to ensure the job can be correctly destroyed later when *remove_links* is called.

Example from `ApsPlanner._schedule`:

```
def _schedule(self, link:Link):
    job = self._scheduler.add_job(link.transfer,
        trigger=IntervalTrigger(seconds=link.interval.total_seconds()))

    link.set_job(job)
```

_unschedule

Unschedule a *Link*. This method runs whenever *remove_links* is called and should not be executed directly. It should accept a link and remove it from the scheduling system you're using. Note that you do not need to remove the link from your planner - BasePlanner will automatically remove that link from *BasePlanner.links* when *remove_links* is called. It isn't required for the scheduling to be already stopped when `_unschedule` is called.

If the scheduler you're using utilises some form of task-managing job objects, you may access these using *Link.job* in order to correctly destroy them if necessary when `_unschedule` is called.

Example from `ApsPlanner._unschedule`:

```
def _unschedule(self, link:Link):
    if link.job is not None:
        link.job.remove()
        link.set_job(None)
```

_start_planner

Start the scheduling. This method runs whenever *BasePlanner.start* is called and should not be executed directly. It should begin the scheduling of links.

This method will be called just after all *Inlet.on_start* and *Outlet.on_start* are called.

Example from `ApsPlanner._start_planner`:

```
def _start_planner(self):
    self._scheduler.start()
```

_shutdown_planner

Shutdown the scheduling. This method runs whenever *BasePlanner.shutdown* is called and should not be executed directly. It should shut down the scheduling of links.

A `wait` parameter is provided that you can pass down to your scheduling system if it allows waiting for the remaining jobs to complete before shutting down.

This method will be called just before all *Inlet.on_shutdown* and *Outlet.on_shutdown* are called.

Example from `ApsPlanner._shutdown_planner`:

```
def _shutdown_planner(self, wait:bool=True):
    self._scheduler.shutdown(wait=wait)
```

Running property

BasePlanner.running property should return a boolean value indicating whether the scheduler is currently running. By default this property always returns True.

Exceptions

When implementing your planner you should consider that links may raise exceptions when executing. Your planner should anticipate this and allow handling the exceptions appropriately to ensure continuous execution. BasePlanner exposes a protected `BasePlanner._on_exception` method that can be called to handle the exception, allowing to ignore exceptions when `ignore_exceptions=True` is passed on construction. Otherwise the exceptions will be logged and the planner will attempt a graceful shutdown. Both *`ApsPlanner`* and *`SchedulePlanner`* support this behaviour by default. See *Exception handling* for more.

Immediate transfer on start

By default BasePlanner will execute *`Link.transfer`* function on all its links once upon calling *`BasePlanner.start`*. This is to avoid having to wait for the link's interval to expire before the first transfer. You can disable this behaviour by passing `immediate_transfer=False` parameter on construction of the *`BasePlanner`* to disable it for all governed links or individually for selected links by setting their `immediate_transfer` to `False`.

Shutdown atexit

Each *`BasePlanner`* registers an `atexit` callback, which will attempt to gracefully shut the planner down if it is created with *`shutdown_at_exit`* parameter set to `True`.

Next Steps

1. Learn about extending *Inlets* and *Outlets*.
2. See the *Examples*

2.3.4 Community Contributions

We aim to support the ecosystem of Databay users by collating and promoting third-party inlets and outlets that implement popular functionalities. We encourage you to share the inlets and outlets you write with the community. See the list of currently shared inlets and outlets, as well as the description of the submission process on Databay's GitHub Page.

Guideline

To ensure your contribution is widely adopted, we recommend the following guideline of implementation.

Read the documentation

Understand the design decisions behind Databay, and the inlets and outlets. Read through the *examples* as well as the currently implemented *`inlets`* and *`outlets`* to understand how Databay can be used.

Write tests

The more reliable your code is, the more likely other users will choose to rely on it. In this StackOverflow question you can read more about why tests matter. You can test some fundamental Databay functionality by using *InletTester*. You should write additional tests outside of scope of *InletTester* to cover the custom logic introduced by you. Remember that apart from writing unit tests, it is easy to write integration tests using *Databay planners*.

See to the tests of the built-in inlets and outlets for reference.

Write documentation

Your inlets and outlets should be well documented. Each implementation will be dependant on the functionality it provides, therefore your design decisions should be laid out and the API explained. We encourage you to write external standalone documentation apart from writing docstrings in code. Your GitHub page should also contain a short introduction, overview and examples.

Correctly use metadata

Inlets

When writing inlets, remember to not modify or read the metadata provided, and to correctly initialise your inlet using `super().__init__(*args, **kwargs)`.

Incorrect:

```
class MyInlet(Inlet):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.my_argument = self.metadata['my_argument']
```

Correct:

```
class MyInlet(Inlet):

    def __init__(self, my_argument, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.my_argument = my_argument
```

Outlets

When writing outlets supporting metadata, you should clearly describe the expected behaviour of each metadata in the documentation.

Your outlet should not exclusively rely on metadata and error out in its absence. Provide a method of setting default values for all metadata you expect and use these when encountering records that don't carry metadata.

To prevent name clashing with other implementations, each metadata key should contain the name of your outlet included in its body.

Incorrect:

```
FILEPATH:MetadataKey = 'FILEPATH'
```

Correct:

```
FILEPATH:MetadataKey = 'CsvOutlet.FILEPATH'
```

2.4 Examples

You can find all examples in the GitHub repository in the Examples folder.

2.4.1 Simple usage

This is a simple example of how data can be produced, transferred and consumed in Databay. It uses built-in *`HttpInlet`* for producing data using a test URL and *`MongoOutlet`* consuming it using MongoDB.

1. Create an inlet for data production:

```
http_inlet = HttpInlet('https://jsonplaceholder.typicode.com/todos/1')
```

1. Create an outlet for data consumption:

```
mongo_outlet = MongoOutlet(database_name='databay',
```

1. Add the two to a link that will handle data transfer between them:

```
link = Link(http_inlet, mongo_outlet,
            datetime.timedelta(seconds=5), tags='http_to_mongo')
```

1. Create a planner, add the link and start scheduling:

```
planner = ApsPlanner(link)
planner.start()
```

1. (Optional) In this example the databay logger is configured to display all messages. See *Logging* for more information.

```
logging.getLogger('databay').setLevel(logging.DEBUG)
```

Output:

```
>>> 2020-07-30 19:51:36.313|I| Added link: Link(tags:['http_to_mongo'],␣
↪inlets:[HttpInlet(metadata:{})], outlets:[MongoOutlet()], interval:0:00:05)␣
↪(databay.BasePlanner)
>>> 2020-07-30 19:51:36.314|I| Starting ApsPlanner(threads:30) (databay.BasePlanner)

>>> 2020-07-30 19:51:41.318|D| http_to_mongo.0 transfer (databay.Link)
>>> 2020-07-30 19:51:41.318|I| http_to_mongo.0 pulling https://jsonplaceholder.
↪typicode.com/todos/1 (databay.HttpInlet)
>>> 2020-07-30 19:51:42.182|I| http_to_mongo.0 received https://jsonplaceholder.
↪typicode.com/todos/1 (databay.HttpInlet)
>>> 2020-07-30 19:51:42.188|I| http_to_mongo.0 insert [{'userId': 1, 'id': 1, 'title
↪': 'delectus aut autem', 'completed': False}] (databay.MongoOutlet)
>>> 2020-07-30 19:51:42.191|I| http_to_mongo.0 written [{'userId': 1, 'id': 1, 'title
↪': 'delectus aut autem', 'completed': False, '_id': ObjectId(
↪'5f22c25ea7aca516ec3fcf38')}] (databay.MongoOutlet)
>>> 2020-07-30 19:51:42.191|D| http_to_mongo.0 done (databay.Link)

>>> 2020-07-30 19:51:46.318|D| http_to_mongo.1 transfer (databay.Link)
>>> 2020-07-30 19:51:46.318|I| http_to_mongo.1 pulling https://jsonplaceholder.
↪typicode.com/todos/1 (databay.HttpInlet)
>>> 2020-07-30 19:51:46.358|I| http_to_mongo.1 received https://jsonplaceholder.
↪typicode.com/todos/1 (databay.HttpInlet)
>>> 2020-07-30 19:51:46.360|I| http_to_mongo.1 insert [{'userId': 1, 'id': 1, 'title
↪': 'delectus aut autem', 'completed': False}] (databay.MongoOutlet)
>>> 2020-07-30 19:51:46.361|I| http_to_mongo.1 written [{'userId': 1, 'id': 1, 'title
↪': 'delectus aut autem', 'completed': False, '_id': ObjectId(
↪'5f22c262a7aca516ec3fcf39')}] (databay.MongoOutlet)
>>> 2020-07-30 19:51:46.362|D| http_to_mongo.1 done (databay.Link)
...
```

Above log can be read as follows:

- At first the planner adds the link provided and starts scheduling:

  ```
  Added link: Link(tags:['http_to_mongo'], inlets:[HttpInlet(metadata:{})],␣
  ↪outlets:[MongoOutlet()], interval:0:00:05)
  Starting ApsPlanner(threads:30)
  ```

- Once scheduling starts, link will log the beginning and end of each transfer:

  ```
  http_to_mongo.0 transfer
  ```

  Note the `http_to_mongo.0` prefix in the message. It is the string representation of the *`Update`* object that represents each individual transfer executed by that particular link. `http_to_mongo` is the tag of the link, while `0` represents the incremental index of the transfer.

- Then *`HttpInlet`* logs its data production:

  ```
  http_to_mongo.0 pulling https://jsonplaceholder.typicode.com/todos/1
  http_to_mongo.0 received https://jsonplaceholder.typicode.com/todos/1
  ```

- Followed by *`MongoOutlet`* logging its data consumption:

  ```
  http_to_mongo.0 insert [{'userId': 1, 'id': 1, 'title': 'delectus aut␣
  ↪autem', 'completed': False}]
  http_to_mongo.0 written [{'userId': 1, 'id': 1, 'title': 'delectus aut
  ```

- Finally, link reports completing its first transfer:

```
http_to_mongo.0 done
```

Full example:

```
import datetime
import logging

from databay import Link
from databay.inlets import HttpInlet
from databay.outlets import MongoOutlet
from databay.planners import ApsPlanner

logging.getLogger('databay').setLevel(logging.DEBUG)

# Create an inlet, outlet and a link.
http_inlet = HttpInlet('https://jsonplaceholder.typicode.com/todos/1')
mongo_outlet = MongoOutlet(database_name='databay',
                           collection='test_collection')
link = Link(http_inlet, mongo_outlet,
            datetime.timedelta(seconds=5), tags='http_to_mongo')

# Create a planner, add the link and start scheduling.
planner = ApsPlanner(link)
planner.start()
```

2.4.2 Basic Inlet

In this example we create a simple implementation of *Inlet*, producing a random integer on a 5 second interval.

1. Extend the *Inlet* class, returning produced data from the *pull* method:

```
class RandomIntInlet(Inlet):

    def pull(self, update):
        return random.randint(0, 100)
```

1. Instantiate it:

```
random_int_inlet = RandomIntInlet()
```

1. Add it to a link:

```
link = Link(random_int_inlet,
            print_outlet,
            interval=timedelta(seconds=5),
            tags='random_ints')
```

1. Add to a planner and schedule.

```
planner = SchedulePlanner(link)
planner.start()
```

Output:

```
>>> random_ints.0 50
>>> random_ints.1 61
```

(continues on next page)

(continued from previous page)

```
>>> random_ints.2 5
>>> ...
```

On each transfer `RandomIntInlet` produces a random integer.

Full example:

```
from databay import Link
from databay.outlets import PrintOutlet
from databay.planners import SchedulePlanner
from datetime import timedelta
from databay import Inlet
import random


class RandomIntInlet(Inlet):

    def pull(self, update):
        return random.randint(0, 100)


random_int_inlet = RandomIntInlet()

print_outlet = PrintOutlet(only_payload=True)

link = Link(random_int_inlet,
            print_outlet,
            interval=timedelta(seconds=5),
            tags='random_ints')

planner = SchedulePlanner(link)
planner.start()
```

2.4.3 Basic Outlet

In this example we create a simple implementation of *Outlet*, printing the incoming records one by one.

1. Extend the *Outlet* class, printing the incoming data in the *push* method:

```
class PrintOutlet(Outlet):

    def push(self, records: [Record], update):
        for record in records:
            print(update, record.payload)
```

1. Instantiate it:

```
print_outlet = PrintOutlet()
```

1. Add it to a link:

```
link = Link(random_int_inlet,
            print_outlet,
            interval=timedelta(seconds=2),
            tags='print_outlet')
```

1. Add to a planner and schedule.

```
planner = SchedulePlanner(link)
planner.start()
```

Output:

```
>>> print_outlet.0 10
>>> print_outlet.1 34
>>> print_outlet.2 18
>>> ...
```

On each transfer `PrintOutlet` prints the payload of records generated by `RandomIntInlet`

Full example:

```
from datetime import timedelta

from databay import Link
from databay.inlets import RandomIntInlet
from databay.planners import SchedulePlanner
from databay.record import Record
from databay.outlet import Outlet


class PrintOutlet(Outlet):

    def push(self, records: [Record], update):
        for record in records:
            print(update, record.payload)


random_int_inlet = RandomIntInlet()
print_outlet = PrintOutlet()

link = Link(random_int_inlet,
            print_outlet,
            interval=timedelta(seconds=2),
            tags='print_outlet')

planner = SchedulePlanner(link)
planner.start()
```

2.4.4 Intermediate Inlet

This example demonstrates an inlet that produces weather prognostic using OpenWeatherMap. It showcases what a realistic implementation of *`Inlet`* may look like.

1. Create the `WeatherInlet` implementing *`Inlet`* class. We expect `api_key` and `city_name` to be provided when constructing this inlet.

```
from databay.inlet import Inlet
import urllib.request


class WeatherInlet(Inlet):
    def __init__(self, api_key: str, city_name: str, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(*args, **kwargs)

        self.api_key = api_key
        self.city_name = city_name
```

1. Implement *pull* method, starting by creating the OpenWeatherMap URL using the `api_key` and `city_name` provided.

```
    def pull(self, update) -> List[Record]:
        url = f'https://api.openweathermap.org/data/2.5/weather?' \
              f'q={self.city_name}&' \
              f'appid={self.api_key}'
```

1. Make a request to OpenWeatherMap using `urllib.request`.

```
        contents = urllib.request.urlopen(url).read().decode('utf8')
```

1. Parse the response and return produced data.

```
        formatted = json.loads(contents)
        return formatted['weather'][0]['description']
```

1. Instantiate `WeatherInlet`.

```
api_key = os.environ.get('OPEN_WEATHER_MAP_API_KEY')
weather_inlet = WeatherInlet(api_key, 'Bangkok')
```

1. Create a link, add it to planner and schedule.

```
link = Link(weather_inlet, PrintOutlet(only_payload=True),
            interval=timedelta(seconds=2), tags='bangkok_weather')

planner = ApsPlanner(link)
planner.start()
```

Output:

```
>>> bangkok_weather.0 light rain
>>> bangkok_weather.1 light rain
>>> bangkok_weather.2 light rain
>>> ...
```

On each transfer `WeatherInlet` makes a request to OpenWeatherMap API and returns a description of the weather in the selected city.

Full example:

```
import json
import os
from datetime import timedelta
from typing import List

from databay import Record, Link
from databay.outlets import PrintOutlet
from databay.planners import ApsPlanner
```

(continues on next page)

(continued from previous page)

```
from databay.inlet import Inlet
import urllib.request


class WeatherInlet(Inlet):
    def __init__(self, api_key: str, city_name: str, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.api_key = api_key
        self.city_name = city_name

    def pull(self, update) -> List[Record]:
        url = f'https://api.openweathermap.org/data/2.5/weather?' \
              f'q={self.city_name}&' \
              f'appid={self.api_key}'

        contents = urllib.request.urlopen(url).read().decode('utf8')

        formatted = json.loads(contents)
        return formatted['weather'][0]['description']


api_key = os.environ.get('OPEN_WEATHER_MAP_API_KEY')
weather_inlet = WeatherInlet(api_key, 'Bangkok')

link = Link(weather_inlet, PrintOutlet(only_payload=True),
            interval=timedelta(seconds=2), tags='bangkok_weather')

planner = ApsPlanner(link)
planner.start()
```

2.4.5 Intermediate Outlet

This example demonstrates an outlet that writes the incoming records into a file. It showcases what a realistic implementation of *Outlet* may look like.

1. Create the `FileOutlet` implementing *Outlet* class. This outlet will accept two metadata keys:
    - `FileOutlet.FILEPATH` - specifying the file that the record should be written into.
    - `FileOutlet.FILE_MODE` - specifying the write mode using Python's default IO.

```
class FileOutlet(Outlet):

    FILEPATH = 'FileOutlet.FILEPATH'
    """Filepath of the file to write to."""

    FILE_MODE = 'FileOutlet.FILE_MODE'
    """Write mode to use when writing into the csv file."""
```

1. We give an option to specify `default_filepath` and `default_file_mode` when constructing this outlet.

```
    def __init__(self,
                 default_filepath: str = 'outputs/default_output.txt',
                 default_file_mode: str = 'a'):
```

(continues on next page)

(continued from previous page)

```
        super().__init__()

        self.default_filepath = default_filepath
        self.default_file_mode = default_file_mode
```

1. Implement `push` method, looping over all records and reading their metadata.

```
    def push(self, records: [Record], update):
        for record in records:
            filepath = record.metadata.get(
                self.FILEPATH, self.default_filepath)
            file_mode = record.metadata.get(
                self.FILE_MODE, self.default_file_mode)
```

1. Write the record according to the `filepath` and `file_mode` found.

```
            with open(filepath, file_mode) as f:
                f.write(str(record.payload)+'\n')
```

1. Instantiate `FileOutlet` and *`RandomIntInlet`* provided with a metadata dictionary.

1. Create a link, add to a planner and schedule.

```
link = Link(random_int_inlet,
            file_outlet,
            interval=timedelta(seconds=2),
            tags='file_outlet')

planner = ApsPlanner(link)
planner.start()
```

Creates `outputs/random_ints.txt` file:

```
1
76
52
76
64
89
71
12
70
74
...
```

Full example:

```
from datetime import timedelta

from databay import Link
from databay.inlets import RandomIntInlet
from databay.planners import ApsPlanner
from databay.record import Record
from databay.outlet import Outlet
```

(continues on next page)

(continued from previous page)

```
class FileOutlet(Outlet):

    FILEPATH = 'FileOutlet.FILEPATH'
    """Filepath of the file to write to."""

    FILE_MODE = 'FileOutlet.FILE_MODE'
    """Write mode to use when writing into the csv file."""

    def __init__(self,
                 default_filepath: str = 'outputs/default_output.txt',
                 default_file_mode: str = 'a'):

        super().__init__()

        self.default_filepath = default_filepath
        self.default_file_mode = default_file_mode

    def push(self, records: [Record], update):
        for record in records:
            filepath = record.metadata.get(
                self.FILEPATH, self.default_filepath)
            file_mode = record.metadata.get(
                self.FILE_MODE, self.default_file_mode)

            with open(filepath, file_mode) as f:
                f.write(str(record.payload)+'\n')


metadata = {
    FileOutlet.FILEPATH: 'outputs/random_ints.txt',
    FileOutlet.FILE_MODE: 'a'
}
random_int_inlet = RandomIntInlet(metadata=metadata)
file_outlet = FileOutlet()

link = Link(random_int_inlet,
            file_outlet,
            interval=timedelta(seconds=2),
            tags='file_outlet')

planner = ApsPlanner(link)
planner.start()
```

2.4.6 Basic metadata

This example demonstrates basic usage of *Global metadata*, used by a PrintOutlet created in the *Basic Outlet* example.

1. Create the `ConditionalPrintOutlet` implementing *`Outlet`* class. This outlet will accept one metadata key:

    - `ConditionalPrintOutlet.SHOULD_PRINT` - whether record should be printed.

```
class ConditionalPrintOutlet(Outlet):

    SHOULD_PRINT = 'ConditionalPrintOutlet.SHOULD_PRINT'
    """Whether records should be printed or skipped."""
```

1. Implement `push` method, looping over all records and printing them if `ConditionalPrintOutlet.SHOULD_PRINT` is set:

```
    def push(self, records: [Record], update):
        for record in records:
            if record.metadata.get(self.SHOULD_PRINT):
                print(update, record)
```

1. Instantiate two inlets, one that always prints, other that never prints:

```
random_int_inlet_on = RandomIntInlet(
    metadata={ConditionalPrintOutlet.SHOULD_PRINT: True})
random_int_inlet_off = RandomIntInlet(
    metadata={ConditionalPrintOutlet.SHOULD_PRINT: False})
```

1. Instantiate `ConditionalPrintOutlet` and add all nodes to a link

```
print_outlet = ConditionalPrintOutlet()

link = Link([random_int_inlet_on, random_int_inlet_off],
            print_outlet,
            interval=timedelta(seconds=0.5),
            tags='should_print_metadata')
```

1. Add to a planner and schedule.

```
planner = SchedulePlanner(link, refresh_interval=0.5)
planner.start()
```

Output:

```
>>> should_print_metadata.0 Record(payload=44, metadata={'PrintOutlet.SHOULD_PRINT':␣
↪True, '__inlet__': "RandomIntInlet(metadata:{'PrintOutlet.SHOULD_PRINT': True})"})
>>> should_print_metadata.1 Record(payload=14, metadata={'PrintOutlet.SHOULD_PRINT':␣
↪True, '__inlet__': "RandomIntInlet(metadata:{'PrintOutlet.SHOULD_PRINT': True})"})
>>> should_print_metadata.2 Record(payload=54, metadata={'PrintOutlet.SHOULD_PRINT':␣
↪True, '__inlet__': "RandomIntInlet(metadata:{'PrintOutlet.SHOULD_PRINT': True})"})
>>> ...
```

On each transfer `ConditionalPrintOutlet` prints records incoming only from the `random_int_inlet_on` that was constructed with global metadata that allows printing.

Full example:

```
from datetime import timedelta

from databay import Link
from databay.inlets import RandomIntInlet
from databay.outlet import Outlet
from databay.planners import SchedulePlanner
from databay.record import Record


class ConditionalPrintOutlet(Outlet):

    SHOULD_PRINT = 'ConditionalPrintOutlet.SHOULD_PRINT'
    """Whether records should be printed or skipped."""
```

(continues on next page)

(continued from previous page)

```
    def push(self, records: [Record], update):
        for record in records:
            if record.metadata.get(self.SHOULD_PRINT):
                print(update, record)


random_int_inlet_on = RandomIntInlet(
    metadata={ConditionalPrintOutlet.SHOULD_PRINT: True})
random_int_inlet_off = RandomIntInlet(
    metadata={ConditionalPrintOutlet.SHOULD_PRINT: False})

print_outlet = ConditionalPrintOutlet()

link = Link([random_int_inlet_on, random_int_inlet_off],
            print_outlet,
            interval=timedelta(seconds=0.5),
            tags='should_print_metadata')

planner = SchedulePlanner(link, refresh_interval=0.5)
planner.start()
```

2.4.7 Basic asynchronous

This tutorial showcases a simple usage of asynchronous inlets and outlets.

1. Create an asynchronous inlet.

```
class RandomIntInlet(Inlet):

    async def pull(self, update):

        # simulate a long-taking operation
        await asyncio.sleep(0.5)

        # execute
        r = random.randint(0, 100)

        _LOGGER.debug(f'{update} produced:{r}')
        return r
```

1. Create an asynchronous outlet. Note that one asynchronous wait will be simulated for each record consumed.

```
class PrintOutlet(Outlet):

    async def push(self, records: [Record], update):
        _LOGGER.debug(f'{update} push starts')

        # create an asynchronous task for each record
        tasks = [self.print_task(record, update) for record in records]

        # await all print tasks
        await asyncio.gather(*tasks)

    async def print_task(self, record, update):
```

(continues on next page)

(continued from previous page)

```
        # simulate a long-taking operation
        await asyncio.sleep(0.5)

        # execute
        _LOGGER.debug(f'{update} consumed:{record.payload}')
```

1. Instantiate three asynchronous inlets and one asynchronous outlet.

```
random_int_inletA = RandomIntInlet()
random_int_inletB = RandomIntInlet()
random_int_inletC = RandomIntInlet()
print_outlet = PrintOutlet()

link = Link([random_int_inletA, random_int_inletB, random_int_inletC],
            print_outlet,
            interval=timedelta(seconds=2),
            tags='async')
```

1. Add to a planner and schedule.

```
planner = SchedulePlanner(link)
planner.start()
```

Output:

```
>>> 2020-08-04 22:40:41.242|D| async.0 transfer
>>> 2020-08-04 22:40:41.754|D| async.0 produced:20
>>> 2020-08-04 22:40:41.754|D| async.0 produced:55
>>> 2020-08-04 22:40:41.754|D| async.0 produced:22
>>> 2020-08-04 22:40:41.755|D| async.0 push starts
>>> 2020-08-04 22:40:42.267|D| async.0 consumed:20
>>> 2020-08-04 22:40:42.267|D| async.0 consumed:55
>>> 2020-08-04 22:40:42.267|D| async.0 consumed:22
>>> 2020-08-04 22:40:42.267|D| async.0 done

>>> 2020-08-04 22:40:43.263|D| async.1 transfer
>>> 2020-08-04 22:40:43.776|D| async.1 produced:10
>>> 2020-08-04 22:40:43.776|D| async.1 produced:4
>>> 2020-08-04 22:40:43.776|D| async.1 produced:90
>>> 2020-08-04 22:40:43.777|D| async.1 push starts
>>> 2020-08-04 22:40:44.292|D| async.1 consumed:10
>>> 2020-08-04 22:40:44.292|D| async.1 consumed:4
>>> 2020-08-04 22:40:44.292|D| async.1 consumed:90
>>> 2020-08-04 22:40:44.292|D| async.1 done
```

On each transfer, two asynchronous operations take place:

- First, all inlets are simultaneously awaiting before producing their data.
- Once all data from inlets is gathered, the second stage commences where the outlet simultaneously awaits for each record before printing it out.

This simulates a delay happening either in the inlets or outlets. Note how one transfer takes approximately a second to complete, despite executing six operations each requiring 0.5 seconds of sleep. If this was to execute synchronously, the entire transfer would take around 3 seconds to complete.

Full example:

```
import asyncio
import logging

from databay import Link, Outlet, Record
from databay.planners import SchedulePlanner
from datetime import timedelta
from databay import Inlet
import random

_LOGGER = logging.getLogger('databay.basic_asynchronous')
logging.getLogger('databay').setLevel(logging.DEBUG)


class RandomIntInlet(Inlet):

    async def pull(self, update):

        # simulate a long-taking operation
        await asyncio.sleep(0.5)

        # execute
        r = random.randint(0, 100)

        _LOGGER.debug(f'{update} produced:{r}')
        return r


class PrintOutlet(Outlet):

    async def push(self, records: [Record], update):
        _LOGGER.debug(f'{update} push starts')

        # create an asynchronous task for each record
        tasks = [self.print_task(record, update) for record in records]

        # await all print tasks
        await asyncio.gather(*tasks)

    async def print_task(self, record, update):

        # simulate a long-taking operation
        await asyncio.sleep(0.5)

        # execute
        _LOGGER.debug(f'{update} consumed:{record.payload}')


random_int_inletA = RandomIntInlet()
random_int_inletB = RandomIntInlet()
random_int_inletC = RandomIntInlet()
print_outlet = PrintOutlet()

link = Link([random_int_inletA, random_int_inletB, random_int_inletC],
            print_outlet,
            interval=timedelta(seconds=2),
            tags='async')
```

(continues on next page)

(continued from previous page)

```
planner = SchedulePlanner(link)
planner.start()
```

2.4.8 Elasticsearch Outlet

In this example we create an implementation of *Outlet* that indexes records as documents to a running Elasticsearch instance.

Note: this example assumes that Elasticsearch is correctly configured and that the index you are indexing documents to exists with the appropriate mappings. For more details see the official Elasticsearch Python client

1. Extend the *Outlet* with new parameters required when constructing: `es_client` - an instance of the elasticsearch python client and `index_name` the name of a pre-existing index in the running cluster.

```
class ElasticsearchIndexerOutlet(Outlet):
    " An example outlet for indexing text documents from any `Inlet`."

    def __init__(self,
                 es_client: elasticsearch.Elasticsearch,
                 index_name: str,
                 overwrite_documents: bool = True):
        super().__init__()
        self.es_client = es_client
        self.index_name = index_name

        # if true existing documents will be overwritten
        # otherwise we will skip indexing and log that document id exists in index.
        self.overwrite_documents = overwrite_documents

        if not self.es_client.indices.exists(self.index_name):
            raise RuntimeError(f"Index '{self.index_name}' does not exist ")
```

1. In this implementation of the `push` method there are a few custom behaviors specified. As we iterate over every incoming record:
    - We use the dict keys from the current record's `payload` as our unique document ID.
    - The flag `self.overwrite_documents` determines whether we will check if an id already exists.
    - If `self.overwrite_documents` is True we simply index the document and `_id` without doing any check.
    - Otherwise we use the client to check if `_id` exists in the index. If it does we skip and log that it already exists. Otherwise it is indexed as normal.

```
    def push(self, records: List[Record], update):
        for record in records:

            payload = record.payload

            # using dict keys from payload as unique id for index
            for k in payload.keys():
                _id = k
                text = payload[k]
                body = {"my_document": text}
                if self.overwrite_documents:
```

(continues on next page)

(continued from previous page)

```
                    self.es_client.index(
                        self.index_name, body, id=_id)
                    _LOGGER.info(f"Indexed document with id {_id}")

                else:
                    if self.es_client.exists(self.index_name, _id):
                        # log that already exists
                        _LOGGER.info(
                            f"Document already exists for id {_id}. Skipping.")
                    else:
                        self.es_client.index(
                            self.index_name, body, id=_id)
                        _LOGGER.info(f"Indexed document with id {_id}")
```

1. This simple *Inlet* takes a list of strings as its main parameter. In its `pull` method it randomly selects one and returns the string and an incrementing id as a `dict`. We'll use this to pass documents to our Elasticsearch Outlet.

```
class DummyTextInlet(Inlet):
    "A simple `Inlet` that randomly pulls a string from a list of strings."

    def __init__(self, text: list, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.text = text
        self._id = 0

    def pull(self, update):
        text_selection = random.choice(self.text)
        self._id += 1
        time.sleep(1)
        return {self._id: text_selection}
```

1. Instantiate our simple *Inlet* as well as an instance of `ElasticsearchIndexerOutlet` with the default parameter for `overwrite_documents`.
    - We use the official Elasticsearch Python client for *es_client*.
    - This example assumes `my-test-index` exists already in our elasticsearch cluster.

```
es_client = elasticsearch.Elasticsearch(timeout=30)

text_inlet = DummyTextInlet(TEXT.split("."))
elasticsearch_outlet = ElasticsearchIndexerOutlet(
    es_client, "my-test-index")
```

1. Tie it all together using *Link* AND `Planner`
    - The link is setup to index a new document every 2 seconds.

```
link = Link(text_inlet,
            elasticsearch_outlet,
            interval=2,
            tags='elasticsearch_outlet')

planner = ApsPlanner(link)
planner.start()
```

Output:

- From the logs we can see that the records are being written into our Elasticsearch index.

```
>>> Indexed document with id 1
>>> Indexed document with id 2
>>> Indexed document with id 3
>>> Indexed document with id 4
>>> Indexed document with id 5
>>> Indexed document with id 6
>>> Indexed document with id 7
>>> Indexed document with id 8
```

Output (if `overwrite_documents` is set to `False`):

- From the logs we can see that the record ID's so far have already been written into our Elasticsearch index.

```
>>> Document already exists for id 1. Skipping.
>>> Document already exists for id 2. Skipping.
>>> Document already exists for id 3. Skipping.
>>> Document already exists for id 4. Skipping.
>>> Document already exists for id 5. Skipping.
>>> Document already exists for id 6. Skipping.
>>> Document already exists for id 7. Skipping.
>>> Document already exists for id 8. Skipping.
```

Full example:

```
import logging
import random
import time
from typing import List

import elasticsearch
from databay import Inlet, Link, Outlet
from databay.planners import ApsPlanner
from databay.record import Record

TEXT = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Phasellus ex erat, viverra tincidunt tempus eget, hendrerit sed ligula.
Quisque mollis nibh in imperdiet porttitor. Nulla bibendum lacus et est lobortis␣
↪porta.
Nulla sed ligula at odio volutpat consectetur. Sed quis augue ac magna porta␣
↪imperdiet interdum eu velit.
Integer pretium ultrices urna, id viverra mauris ultrices ut. Etiam aliquet tellus␣
↪porta nisl eleifend, non hendrerit nisl sodales.
Aliquam eget porttitor enim.
"""

_LOGGER = logging.getLogger('databay.elasticsearch_outlet')


class ElasticsearchIndexerOutlet(Outlet):
    " An example outlet for indexing text documents from any `Inlet`."

    def __init__(self,
                 es_client: elasticsearch.Elasticsearch,
                 index_name: str,
```

(continues on next page)

(continued from previous page)

```
                 overwrite_documents: bool = True):
        super().__init__()
        self.es_client = es_client
        self.index_name = index_name

        # if true existing documents will be overwritten
        # otherwise we will skip indexing and log that document id exists in index.
        self.overwrite_documents = overwrite_documents

        if not self.es_client.indices.exists(self.index_name):
            raise RuntimeError(f"Index '{self.index_name}' does not exist ")

    def push(self, records: List[Record], update):
        for record in records:

            payload = record.payload

            # using dict keys from payload as unique id for index
            for k in payload.keys():
                _id = k
                text = payload[k]
                body = {"my_document": text}
                if self.overwrite_documents:
                    self.es_client.index(
                        self.index_name, body, id=_id)
                    _LOGGER.info(f"Indexed document with id {_id}")

                else:
                    if self.es_client.exists(self.index_name, _id):
                        # log that already exists
                        _LOGGER.info(
                            f"Document already exists for id {_id}. Skipping.")
                    else:
                        self.es_client.index(
                            self.index_name, body, id=_id)
                        _LOGGER.info(f"Indexed document with id {_id}")


class DummyTextInlet(Inlet):
    "A simple `Inlet` that randomly pulls a string from a list of strings."

    def __init__(self, text: list, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.text = text
        self._id = 0

    def pull(self, update):
        text_selection = random.choice(self.text)
        self._id += 1
        time.sleep(1)
        return {self._id: text_selection}


_LOGGER.setLevel(logging.INFO)

es_client = elasticsearch.Elasticsearch(timeout=30)
```

(continues on next page)

(continued from previous page)

```
text_inlet = DummyTextInlet(TEXT.split("."))
elasticsearch_outlet = ElasticsearchIndexerOutlet(
    es_client, "my-test-index")

link = Link(text_inlet,
            elasticsearch_outlet,
            interval=2,
            tags='elasticsearch_outlet')

planner = ApsPlanner(link)
planner.start()
```

2.4.9 Twitter Inlet

In this example we create an implementation of an *Inlet* that connects to the Twitter API and either listens for new tweets for a specific user or to the home timeline for an authenticated account.

Note: this example assumes that the Tweepy client is correctly configured and that the Twitter account is registered to use the API. For more details on Tweepy click here.

1. Extend the *Inlet* by passing in an instance of the Tweepy client `api`. Depending on the use case users can also pass in `user` if they want to run the Inlet on a specific username.

```
class TwitterInlet(Inlet):
    """
    An implementation of an `Inlet` that uses the Tweepy (https://www.tweepy.org/)
    Twitter client to pull tweets from either a specific users' timeline or the
    home timeline belonging to an authenticated `tweepy.API` instance.
    """

    def __init__(self, api: tweepy.API, user: str = None, most_recent_id=None, *args,␣
↪**kwargs):
        super().__init__(*args, **kwargs)
        self.api = api
        self.user = user

        # this will ensure we only every pull tweets that haven't been handled
        self.most_recent_id = most_recent_id

        # sets flag indicating whether we are pulling from as single user
        # or from the home timeline.
        if self.user is None:
            self.is_user_timeline = False
        else:
            self.is_user_timeline = True
```

1. For the `pull` method we perform a number of configuration specific checks:
   - If the flag `self.is_user_timeline` is `True` we'll be using the `user_timeline` method of the Tweepy API. This pulls tweets from a specific users' timeline rather than the registered accounts' home timeline.
   - Additionally there is a check in both conditional branches that checks for `self.most_recent_id`, if a recent ID exists then this ID is passed an additional parameter to Tweepy. This will ensure that only new tweets since the last pull are fetched.
   - `self.most_recent_id` is assigned by taking the ID from the first tweet in the results list.

```
    def pull(self, update):
        if self.is_user_timeline:
            if self.most_recent_id is not None:
                public_tweets = self.api.user_timeline(
                    self.user, since_id=self.most_recent_id)
            else:
                public_tweets = self.api.user_timeline(
                    self.user)
        else:
            if self.most_recent_id is not None:
                public_tweets = self.api.home_timeline(
                    since_id=self.most_recent_id)
            else:
                public_tweets = self.api.home_timeline()

        if len(public_tweets) > 0:
            # 0th tweet is most recent
            self.most_recent_id = public_tweets[0].id

        tweets = []
        for tweet in public_tweets:
            tweets.append({"user": tweet.user.screen_name, "text": tweet.text})
        return tweets
```

1. To authenticate Tweepy correctly the appropriate keys and secrets must be passed to the API.

```
auth = tweepy.OAuthHandler(
    consumer_key, consumer_secret)  # user defined values
auth.set_access_token(access_token, access_token_secret)  # user defined values

# extra params here protect against twitter rate limiting
# set link intervals with this in mind
# for more on twitter rate limiting see https://developer.twitter.com/en/docs/rate-
↪limits
api = tweepy.API(auth, wait_on_rate_limit=True, wait_on_rate_limit_notify=True)
```

1. The `TwitterInlet` can then be instantiated as seen below. We are using the *`PrintOutlet`* to print the results of each successful pull.

    - **Note**: Be mindful of the `interval` you pass to *`Link`* as the Twitter API has strict rate limiting policies.

```
# create TwitterUserInlet() pointed at a specific account name
twitter_user_inlet = TwitterInlet(api, "@BarackObama")

link = Link(twitter_user_inlet, PrintOutlet(only_payload=True),
            interval=30, tags='twitter_timeline')

planner = SchedulePlanner(link)
planner.start()
```

Output:

```
>>> {'user': 'BarackObama', 'text': 'Georgia's runoff election will determine whether␣
↪the American people have a Senate that's actually fighting for the... https://t.co/
↪igUiRzxNxe'}
>>> {'user': 'BarackObama', 'text': 'Here's a great way to call voters in Georgia and␣
↪help them get ready to vote. A couple hours this weekend could hel... https://t.co/
↪x6Nc8w7F38'}
```

(continues on next page)

(continued from previous page)

```
>>> {'user': 'BarackObama', 'text': "Happy Hanukkah to all those celebrating around␣
↪the world. This year has tested us all, but it's also clarified what... https://t.
↪co/k2lzUQ9LNm"}
>>> {'user': 'BarackObama', 'text': 'In A Promised Land, I talk about the decisions I␣
↪had to make during the first few years of my presidency. Here are... https://t.co/
↪KbE2FDStYr'}
>>> {'user': 'BarackObama', 'text': "With COVID-19 cases reaching an all-time high␣
↪this week, we've got to continue to do our part to protect one anothe... https://t.
↪co/Gj0mEFfuLY"}
>>> {'user': 'BarackObama', 'text': 'To all of you in Georgia, today is the last day␣
↪to register to vote in the upcoming runoff election. Take a few min... https://t.co/
↪Jif3Gd7NpQ'}
```

Full example:

```
import os

import tweepy
from databay import Inlet, Link
from databay.outlets import PrintOutlet
from databay.planners import SchedulePlanner


class TwitterInlet(Inlet):
    """
    An implementation of an `Inlet` that uses the Tweepy (https://www.tweepy.org/)
    Twitter client to pull tweets from either a specific users' timeline or the
    home timeline belonging to an authenticated `tweepy.API` instance.
    """

    def __init__(self, api: tweepy.API, user: str = None, most_recent_id=None, *args,␣
↪**kwargs):
        super().__init__(*args, **kwargs)
        self.api = api
        self.user = user

        # this will ensure we only every pull tweets that haven't been handled
        self.most_recent_id = most_recent_id

        # sets flag indicating whether we are pulling from as single user
        # or from the home timeline.
        if self.user is None:
            self.is_user_timeline = False
        else:
            self.is_user_timeline = True

    def pull(self, update):
        if self.is_user_timeline:
            if self.most_recent_id is not None:
                public_tweets = self.api.user_timeline(
                    self.user, since_id=self.most_recent_id)
            else:
                public_tweets = self.api.user_timeline(
                    self.user)
        else:
            if self.most_recent_id is not None:
                public_tweets = self.api.home_timeline(
```

(continues on next page)

(continued from previous page)

```
                    since_id=self.most_recent_id)
            else:
                public_tweets = self.api.home_timeline()

        if len(public_tweets) > 0:
            # 0th tweet is most recent
            self.most_recent_id = public_tweets[0].id

        tweets = []
        for tweet in public_tweets:
            tweets.append({"user": tweet.user.screen_name, "text": tweet.text})
        return tweets


# gets twitter api secrets and keys from environment vars
consumer_key = os.getenv("twitter_key")
consumer_secret = os.getenv("twitter_secret")
access_token = os.getenv("twitter_access_token")
access_token_secret = os.getenv("twitter_access_token_secret")


auth = tweepy.OAuthHandler(
    consumer_key, consumer_secret)  # user defined values
auth.set_access_token(access_token, access_token_secret)  # user defined values

# extra params here protect against twitter rate limiting
# set link intervals with this in mind
# for more on twitter rate limiting see https://developer.twitter.com/en/docs/rate-
↪limits
api = tweepy.API(auth, wait_on_rate_limit=True, wait_on_rate_limit_notify=True)


# create TwitterUserInlet() pointed at a specific account name
twitter_user_inlet = TwitterInlet(api, "@BarackObama")

link = Link(twitter_user_inlet, PrintOutlet(only_payload=True),
            interval=30, tags='twitter_timeline')

planner = SchedulePlanner(link)
planner.start()
```

2.5 Databay API Reference

2.5.1 databay.inlets

databay.inlets.file_inlet

Contents:

FileInletMode	Enum defining the mode in which the FileInlet should read the file.
FileInlet	Inlet producing data by reading a file.

class databay.inlets.file_inlet.**FileInletMode**
databay.inlets.file_inlet.FileInletMode

Enum defining the mode in which the FileInlet should read the file.

Create and return a new object. See help(type) for accurate signature.

Bases: enum.Enum

LINE :str = line
Read file one line per transfer. This will open the file and hold it open for as long as the planner is running.

FILE :str = file
Read the entire file on each transfer. This will only open the file briefly during the transfer.

class databay.inlets.file_inlet.**FileInlet**(*filepath: str*, *read_mode: FileInletMode = FileInletMode.LINE*, **args*, ***kwargs*)
databay.inlets.file_inlet.FileInlet

Inlet producing data by reading a file.

Parameters

- **filepath** (*str*) – Path to the file.
- **read_mode** (*FileInletMode*) – Mode in which the file is to be read.

Bases: databay.Inlet

pull(*self*, *update*)
Produce data by reading a file in the mode specified.

Raises FileNotFoundError if file does not exists.

Returns contents of the file.

on_start(*self*)
If read mode is *FileInletMode.LINE*, open the file and hold it open for reading.

Raises FileNotFoundError if file does not exists.

on_shutdown(*self*)
If read mode is *FileInletMode.LINE*, close the file.

databay.inlets.http_inlet

Warning: *HttpInlet* requires AIOHTTP to function. Please install required dependencies using:

```
pip install "databay[HttpInlet]"
```

class databay.inlets.http_inlet.**HttpInlet**(*url: str*, *json: str = True*, *cacert: Optional[str] = None*, *params: Optional[dict] = None*, *headers: Optional[LooseHeaders] = None*, **args*, ***kwargs*)
databay.inlets.http_inlet.HttpInlet

Inlet for pulling data from a specified URL using aiohttp.

Parameters

- **url** (*str*) – URL that should be queried for data.

- **json** (*bool*) – Whether response should be parsed as JSON. `True`
- **cacert** (*str*) – Path to cacert TLS certificate bundle. `None`
- **params** (*dict*) – Parameters for the request. `None`
- **headers** (*LooseHeaders*) – Headers for the request. `None`

Bases: *databay.inlet.Inlet*

pull(*self*, *update*) → Union[List[Record], str]
async

Asynchronously pulls data from the specified URL using aiohttp.ClientSession.get

Parameters **update** (*Update*) – Update object representing the particular Link transfer.

Returns Single or multiple records produced.

Return type *Record* or list[*Record*]

databay.inlets.null_inlet

class databay.inlets.null_inlet.**NullInlet**(*metadata: dict = None*)
databay.inlets.null_inlet.NullInlet

Inlet that doesn't do anything, essentially a 'no-op' inlet.

Parameters **metadata** (*dict*) – Global metadata that will be attached to each record generated by this inlet. It can be overridden or appended to by providing metadata when creating a record using `new_record()` function. `None`

Bases: `databay.Inlet`

pull(*self*, *update*)
Doesn't produce anything.

Returns empty list

databay.inlets.random_int_inlet

class databay.inlets.random_int_inlet.**RandomIntInlet**(*min: int = 0*, *max: int = 100*, **args*, ***kwargs*)
databay.inlets.random_int_inlet.RandomIntInlet

Inlet that will generate a random integer within the specified range.

Parameters

- **min** (*int*) – Lower boundary of the random range.
- **max** (*int*) – Upper boundary of the random range.

Bases: `databay.Inlet`

pull(*self*, *update*)
Produces a random integer within the specified range.

Parameters **update** (*Update*) – Update object representing the particular Link update run.

Returns Single or multiple records produced.

Return type *Record* or list[*Record*]

2.5.2 databay.misc

databay.misc.inlet_tester

Contents:

for_each_inlet	Runs the test for each inlet returned from *InletTester.get_inlet*
InletTester	Utility class used for testing concrete implementations of *Inlet*.

databay.misc.inlet_tester.**for_each_inlet**(*fn*)
Runs the test for each inlet returned from *InletTester.get_inlet*

class databay.misc.inlet_tester.**InletTester**(*methodName='runTest'*)
databay.misc.inlet_tester.InletTester

Utility class used for testing concrete implementations of *Inlet*.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Bases: unittest.TestCase

get_inlet(*self*)
Implement this method to return instances of your inlet class.

setUp(*self*)
Hook method for setting up the test fixture before exercising it.

test_new_record(*self*)
for_each_inlet

Test creating new records and passing local metadata.

test_new_record_override_global(*self*)
for_each_inlet

Test creating new records and overriding global metadata.

test_dont_read_metadata(*self*, *update*)
for_each_inlet

Test creating new records and overriding global metadata.

test_pull(*self*, *update*)
for_each_inlet

Test pulling data from the inlet.

2.5.3 databay.outlets

databay.outlets.csv_outlet

class databay.outlets.csv_outlet.**CsvOutlet**(*default_filepath: str*, *default_file_mode: str = 'a'*, **args*, ***kwargs*)

databay.outlets.csv_outlet.CsvOutlet

Outlet that writes records to a csv file.

Parameters

- **default_filepath** (*str*) – Filepath of the default csv file to write records to.
- **default_file_mode** (*str*) – Default write mode to use when writing into the csv file.

Bases: *databay.outlet.Outlet*

Record metadata supported:

CSV_FILE :MetadataKey = CsvOutlet.CSV_FILE
Filepath of the csv file to write records to.

FILE_MODE :MetadataKey = CsvOutlet.FILE_MODE
Write mode to use when writing into the csv file.

push(*self*, *records: [Record]*, *update*)
Writes records to a csv file.

Parameters

- **records** (list[*Record*]) – List of records generated by inlets. Each top-level element of this array corresponds to one inlet that successfully returned data. Note that inlets could return arrays too, making this a nested array.
- **update** (*Update*) – Update object representing the particular Link transfer.

databay.outlets.file_outlet

class databay.outlets.file_outlet.**FileOutlet**(*default_filepath: str*, *default_file_mode: str = 'a'*, *default_encoding: str = 'utf-8'*, **args*, ***kwargs*)

databay.outlets.file_outlet.FileOutlet

Outlet that writes records to a file.

Parameters

- **default_filepath** (*str*) – Filepath of the default file to write records to.
- **default_file_mode** (*str*) – Default write mode to use when writing into the file.
- **default_encoding** (*str*) – Default file encoding when writing into a file. `utf-8`

Bases: *databay.outlet.Outlet*

Record metadata supported:

FILEPATH :MetadataKey = FileOutlet.FILEPATH
Filepath of the file to write to.

FILE_MODE :MetadataKey = FileOutlet.FILE_MODE
Write mode to use when writing into the file.

FILE_ENCODING :MetadataKey = FileOutlet.FILE_ENCODING
Encoding to use when writing into the file.

push (*self*, *records: [Record]*, *update*)
Writes records to a file.

Parameters

- **records** (list[*Record*]) – List of records generated by inlets. Each top-level element of this array corresponds to one inlet that successfully returned data. Note that inlets could return arrays too, making this a nested array.
- **update** (*Update*) – Update object representing the particular Link transfer.

databay.outlets.mongo_outlet

Warning: *MongoOutlet* requires PyMongo to function. Please install required dependencies using:

```
pip install "databay[MongoOutlet]"
```

Contents:

MongoCollectionNotFound	Raised when requested collection does not exist in the database.
ensure_connection	Ensure the MongoDB connection is established before running the function.
MongoOutlet	Outlet for pushing data into a MongoDB instance. Pushes are executed synchronously.

exception databay.outlets.mongo_outlet.**MongoCollectionNotFound**
databay.outlets.mongo_outlet.MongoCollectionNotFound

Raised when requested collection does not exist in the database.

Initialize self. See help(type(self)) for accurate signature.

Bases: Exception

databay.outlets.mongo_outlet.**ensure_connection**(*fn*)
Ensure the MongoDB connection is established before running the function.

Parameters **fn** (Callable) – Function to decorate

class databay.outlets.mongo_outlet.**MongoOutlet**(*database_name: str = 'databay'*, *collection: str = 'default_collection'*, *host: str = None*, *port: str = None*, **args*, ***kwargs*)
databay.outlets.mongo_outlet.MongoOutlet

Outlet for pushing data into a MongoDB instance. Pushes are executed synchronously.

Parameters

- **database_name** (*str*) – Name of the MongoDB database to write to. 'databay'
- **collection** (*str*) – Global name of the collection to write to. This can be overwritten by records' metadata.MONGODB_COLLECTION parameter. 'default_collection'

- **host** (*str*) – Address of MongoDB host. `None` (PyMongo defaults to `'localhost'`)
- **port** (*int*) – Port of the MongoDB host. `None` (PyMongo defaults to `27017`)

Bases: *databay.outlet.Outlet*

Record metadata supported:

MONGODB_COLLECTION :MetadataKey = MongoOutlet.MONGODB_COLLECTION
Name of collection to write to.

push(*self*, *records: [Record]*, *update*)
ensure_connection

Write records into the database. Writes are executed synchronously.

Parameters

- **records** (list[*Record*]) – List of records generated by inlets. Each top-level element of this array corresponds to one inlet that successfully returned data. Note that inlets could return arrays too, making this a nested array.
- **update** (*Update*) – Update object representing the particular Link transfer.

connect(*self*, *database_name: str = None*) → bool
Connect to the specified database. Returns True if already connected to the specified database. Disconnects from any existing databases if the specified database is different.

Parameters **database_name** (*str*) – Name of the database to connect to. `None` (Connects to default database name if not specified`)

Returns Returns True if already connected to the database specified.

Return type bool

disconnect(*self*)
Disconnect from the database if currently connected.

on_start(*self*)
Connect to the MongoDB host on start.

on_shutdown(*self*)
Disconnect from the MongoDB host on shutdown.

databay.outlets.null_outlet

class databay.outlets.null_outlet.**NullOutlet**(*processors: Union[callable, List[callable]] = None*)

databay.outlets.null_outlet.NullOutlet

Outlet that doesn't do anything, essentially a 'no-op' outlet.

Parameters **processors** (`callable` or list[`callable`]) – *Processors* of this outlet. `None`

Bases: *databay.outlet.Outlet*

push(*self*, *records: [Record]*, *update*)
async

Doesn't do anything.

databay.outlets.print_outlet

class databay.outlets.print_outlet.**PrintOutlet**(*only_payload: bool = False*, *skip_update: bool = False*, **args*, ***kwargs*)

databay.outlets.print_outlet.PrintOutlet

Outlet that will print all records one by one.

Parameters

- **only_payload** (*bool*) – If True, prints only the payload of records.
- **skip_update** (*bool*) – If True, Update prefix will not be added to the print.

Bases: *databay.outlet.Outlet*

push(*self*, *records: [Record]*, *update*)
async

Prints the records.

Parameters

- **records** (list[*Record*]) – List of records generated by inlets. Each top-level element of this array corresponds to one inlet that successfully returned data. Note that inlets could return arrays too, making this a nested array.
- **update** (*Update*) – Update object representing the particular Link update run.

2.5.4 databay.planners

databay.planners.aps_planner

See also:

- *Scheduling* to learn more about scheduling in Databay.
- *BasePlanner* for the remaining interface of this planner.

Contents:

ApsPlanner	Planner implementing scheduling using the Advanced Python Scheduler. Scheduling sets the `APS Job` as links' job.
APSPlanner	Planner implementing scheduling using the Advanced Python Scheduler. Scheduling sets the `APS Job` as links' job.

class databay.planners.aps_planner.**ApsPlanner**(*links: Union[Link, List[Link]] = None*, *threads: int = 30*, *executors_override: dict = None*, *job_defaults_override: dict = None*, *ignore_exceptions: bool = False*, *catch_exceptions: bool = None*, *immediate_transfer: bool = True*)

databay.planners.aps_planner.ApsPlanner

Planner implementing scheduling using the Advanced Python Scheduler. Scheduling sets the `APS Job` as links' job.

Parameters

- **links** (*Link* or list[*Link*]) – Links that should be added and scheduled. None
- **threads** (*int*) – Number of threads available for job execution. Each link will be run on a separate thread job. 30
- **executors_override** (*dict*) – Overrides for executors option of APS configuration None
- **job_defaults_override** (*dict*) – Overrides for job_defaults option of APS configuration None
- **ignore_exceptions** (*bool*) – Whether exceptions should be ignored or halt the planner. False
- **immediate_transfer** (bool) – Whether planner should execute one transfer immediately upon starting. True

Bases: *databay.base_planner.BasePlanner*

start (*self*)
Start this planner. Calls APS Scheduler.start()

See *Start and Shutdown* to learn more about starting and shutdown.

pause (*self*)
Pause this planner. Calls APScheduler.pause()

resume (*self*)
Resume this planner. Calls APScheduler.resume()

shutdown (*self*, *wait: bool = True*)
Shutdown this planner. Calls APScheduler.shutdown()

See *Start and Shutdown* to learn more about starting and shutdown.

Parameters **wait** (*bool*) – Whether to wait until all currently executing jobs have finished. True

purge (*self*)
Unschedule and clear all links. It can be used while planner is running. APS automatically removes jobs, so we only clear the links.

running (*self*)
property

Whether this planner is currently running. Changed by calls to *start* and *shutdown*.

Returns State of this planner

Return type bool

class databay.planners.aps_planner.**APSPlanner**(**args*, ***kwargs*)
databay.planners.aps_planner.APSPlanner

Planner implementing scheduling using the Advanced Python Scheduler. Scheduling sets the APS Job as links' job.

Parameters

- **links** (*Link* or list[*Link*]) – Links that should be added and scheduled. None
- **threads** (*int*) – Number of threads available for job execution. Each link will be run on a separate thread job. 30

- **executors_override** (*dict*) – Overrides for executors option of APS configuration None
- **job_defaults_override** (*dict*) – Overrides for job_defaults option of APS configuration None
- **ignore_exceptions** (*bool*) – Whether exceptions should be ignored or halt the planner. False
- **immediate_transfer** (bool) – Whether planner should execute one transfer immediately upon starting. True

Bases: *databay.planners.aps_planner.ApsPlanner*

databay.planners.schedule_planner

See also:

- *Scheduling* to learn more about scheduling in Databay.
- *BasePlanner* for the remaining interface of this planner.

Contents:

ScheduleIntervalError	Raised when link interval is smaller than the Schedule refresh interval.
SchedulePlanner	Planner implementing scheduling using Schedule. Scheduling sets the Schedule's Job as links' job.

exception databay.planners.schedule_planner.**ScheduleIntervalError**

databay.planners.schedule_planner.ScheduleIntervalError

Raised when link interval is smaller than the Schedule refresh interval.

Initialize self. See help(type(self)) for accurate signature.

Bases: Exception

class databay.planners.schedule_planner.**SchedulePlanner**(*links: Union[Link, List[Link]] = None*, *threads: int = 30*, *refresh_interval: float = 1.0*, *ignore_exceptions: bool = False*, *catch_exceptions: bool = None*, *immediate_transfer: bool = True*)

databay.planners.schedule_planner.SchedulePlanner

Planner implementing scheduling using Schedule. Scheduling sets the Schedule's Job as links' job.

Parameters

- **links** (*Link* or list[*Link*]) – Links that should be added and scheduled. None
- **threads** (int) – Number of threads to use. 30
- **refresh_interval** (float) – Frequency at which this planner will scan over its links and attempt to update them if necessary. Note that adding links with intervals smaller than this value will raise a *ScheduleIntervalError*. 1.0

- **ignore_exceptions** (bool) – Whether exceptions should be ignored, or halt the planner. False
- **immediate_transfer** (bool) – Whether planner should execute one transfer immediately upon starting. True

Bases: *databay.base_planner.BasePlanner*

refresh_interval (*self*) → float
property

Frequency at which this planner will scan over its links and attempt to update them if necessary. Note that adding links with interval smaller than this value will raise a *ScheduleIntervalError*.

Returns Refresh interval frequency.

Return type float

start (*self*)
Start this planner. Links will start being scheduled based on their intervals after calling this method. Creates a new thread pool if one doesn't already exist.

See *Start and Shutdown* to learn more about starting and shutdown.

shutdown (*self*, *wait: bool = True*)
Stop this planner. Links will stop being scheduled after calling this method

See *Start and Shutdown* to learn more about starting and shutdown.

Parameters **wait** (*bool*) – Whether to wait until all currently executing jobs have finished. True

running (*self*)
property

Whether this planner is currently running. If there are links transferring this may be set before all transfers are complete. Changed by calls to *start* and *shutdown*.

Returns State of this planner

Return type bool

2.5.5 databay.support

databay.support.buffers

class databay.support.buffers.**Buffer**(*count_threshold: int = None*, *time_threshold: float = None*, *custom_controllers: Union[callable, List[callable]] = None*, *on_reset: callable = None*, *conjugate_controllers: bool = False*)
databay.support.buffers.Buffer

Buffers are special built-in *Processors*. They allow you to temporarily accumulate records before passing them over to outlets.

When processing records (see *Processors*) a Buffer will figure out whether records should be stored or released. This is done by passing the list of records to Buffer's internal callable functions called controllers.

Each controller performs different types of checks, returning True or False depending on whether records should be released or stored respectively.

Parameters

- **count_threshold** (*int*) – The number of records stored that when reached will complete the count controller. When set to `None` it will disable the count controller. `None`
- **time_threshold** (*float*) – The number of seconds elapsed since the previous release that when reached will complete the time controller. When set to `None` it will disable the time controller. `None`
- **custom_controllers** (`callable` or list[`callable`]) – List of custom `callable` controllers. `None`
- **on_reset** (`callable`) – Callback invoked when *reset* is called. `None`
- **conjugate_controllers** (*bool*) – Whether to release the records when any controller returns `True` or to wait for all of them to complete before releasing records. `False`

get_controllers (*self*)
Return the list of currently active controllers.

Returns list of controllers

Return type list[`callable`]

reset (*self*)
Resets this buffer, resetting the controllers' counters and emptying the list of records stored.

2.5.6 databay.base_planner

See also:

Extending BasePlanner to learn how to extend this class correctly.

class databay.base_planner.**BasePlanner**(*links: Union[Link, List[Link]] = None*, *ignore_exceptions: bool = False*, *immediate_transfer: bool = True*, *shutdown_at_exit: bool = False*)
databay.base_planner.BasePlanner

Base abstract class for a job planner. Implementations should handle scheduling link transfers based on `datetime.timedelta` intervals.

Parameters

- **links** (*Link* or list[*Link*]) – Links that should be added and scheduled.
- **ignore_exceptions** (`bool`) – Whether exceptions should be ignored, or halt the planner. `False`
- **immediate_transfer** (`bool`) – Whether this planner should execute transfer once immediately upon starting for all links that have `Link.immediate_transfer` set to `True`. `True`
- **shutdown_at_exit** (*bool*) – Whether this planner should attempt to gracefully shutdown if the app exists unexpectedly. `False`

Bases: `abc.ABC`

links (*self*)
property

Links currently handled by this planner.

Returns list[*Link*]

add_links (*self*, *links: Union[Link, List[Link]]*)
Add new links to this planner. This can be run once planner is already running.

Parameters **links** (*Link* or list[*Link*]) – Links that should be added and scheduled.

remove_links (*self*, *links: Link*)
Remove links from this planner. This can be run once planner is already running.

Parameters **links** (*Link* or list[*Link*]) – Links that should be unscheduled and removed.

Raises MissingLinkError if link is not found.

start (*self*)
Start this planner. Links will start being scheduled based on their intervals after calling this method. The exact methodology depends on the planner implementation used.

This will also loop over all links and call the on_start callback before starting the planner.

If BasePlanner.immediate_transfer is set to True, this function will additionally call *Link.transfer* once for each link managed by this planner before starting.

See *Start and Shutdown* to learn more about starting and shutdown.

shutdown (*self*, *wait: bool = True*)
Shutdown this planner. Links will stop being scheduled after calling this method. Remaining link jobs may still execute after calling this method depending on the concrete planner implementation.

This will also loop over all links and call the on_shutdown callback after shutting down the planner.

See *Start and Shutdown* to learn more about starting and shutdown.

purge (*self*)
Unschedule and clear all links. It can be used while planner is running.

running (*self*)
property

Whether this planner is currently running.

By default always returns True.

Override this property to indicate when the underlying scheduling functionality is currently running.

force_transfer (*self*)
Immediately force a transfer on all Links governed by this planner.

2.5.7 databay.errors

Contents:

MissingLinkError	Raised when providing a link that isn't stored in planner.
ImplementationError	Raised when concrete implementation is incorrect.
InvalidNodeError	Raised when invalid node (inlet or outlet) is provided.

exception databay.errors.**MissingLinkError**
databay.errors.MissingLinkError

Raised when providing a link that isn't stored in planner.

Initialize self. See help(type(self)) for accurate signature.

Bases: RuntimeError

exception databay.errors.**ImplementationError**
databay.errors.ImplementationError

Raised when concrete implementation is incorrect.

Initialize self. See help(type(self)) for accurate signature.

Bases: RuntimeError

exception databay.errors.**InvalidNodeError**
databay.errors.InvalidNodeError

Raised when invalid node (inlet or outlet) is provided.

Initialize self. See help(type(self)) for accurate signature.

Bases: RuntimeError

2.5.8 databay.inlet

See also:

- *Extending Inlets* to learn how to extend this class correctly.
- *Outlet* representing the corresponding output of the data stream.

class databay.inlet.**Inlet**(*metadata: dict = None*)
databay.inlet.Inlet

Abstract class representing an input of the data stream.

Parameters **metadata** (*dict*) – Global metadata that will be attached to each record generated by this inlet. It can be overridden or appended to by providing metadata when creating a record using *new_record()* function. None

Bases: abc.ABC

metadata(*self*)
property

Global metadata that will be attached to each record generated by this inlet. It can be overridden or appended to by providing metadata when creating a record using *new_record()* function.

Returns Metadata dictionary.

Return type dict

pull(*self*, *update: da.Update*) → List[Record]
abstractmethod

Produce new data.

Override this method to define how this inlet will produce new data.

Parameters **update** (*Update*) – Update object representing the particular Link update run.

Returns List of records produced

Return type list[*Record*]

new_record(*self*, *payload*, *metadata: dict = None*) → Record
Create a new *Record*. This should be the preferred way of creating new records.

Parameters

- **payload** (*Any*) – Data produced by this inlet.
- **metadata** (*dict*) – Local metadata that will override and/or append to the global metadata. It will be attached to the new record. None

Returns New record created

Return type *Record*

try_start (*self*)
Wrapper around on_start call that will ensure it only gets executed once.

on_start (*self*)
Called once per inlet just before the governing planner is about to start.

Override this method to provide starting functionality on this inlet.

try_shutdown (*self*)
Wrapper around on_shutdown call that will ensure it only gets executed once.

on_shutdown (*self*)
Called once per inlet just after the governing planner has shutdown.

Override this method to provide shutdown functionality on this inlet.

active (*self*)
property

Whether this inlet is active and ready to pull. This variable is set by the governing link to `True` on start and to `False` on shutdown. `False`

Return type bool

2.5.9 databay.link

Contents:

Update	Data structure representing one Link transfer. When converted to string returns `{tags}.{transfer_number}`
Link	Link in the relationship graph. Use this class to define relationships between inlets and outlets.

class databay.link.**Update**(*tags: List[str]*, *transfer_number: int*)
databay.link.Update

Data structure representing one Link transfer. When converted to string returns `{tags}.{transfer_number}`

Parameters

- **tags** (*List[str]*) – Tags of the link, see: *Link*.
- **transfer_number** (*int*) – Incremental identifier of the current transfer.

class databay.link.**Link**(*inlets: Union[Inlet, List[Inlet]]*, *outlets: Union[Outlet, List[Outlet]]*, *interval: Union[datetime.timedelta, int, float]*, *tags: Union[str, List[str]] = None*, *copy_records: bool = True*, *ignore_exceptions: bool = False*, *catch_exceptions: bool = None*, *inlet_concurrency: int = 9999*, *immediate_transfer: bool = True*, *processors: Union[callable, List[callable]] = None*, *groupers: Union[callable, List[callable]] = None*, *name=None*)
databay.link.Link

Link in the relationship graph. Use this class to define relationships between inlets and outlets.

Parameters

- **inlets** (*Inlet* or list[*Inlet*]) – inlets to add to this link.

- **outlets** (*Outlet* or list[*Outlet*]) – outlets to add to this link.
- **interval** (*Union[datetime.timedelta, int, float]*) – Expects `datetime.timedelta`. Alternatively, you can provide `int` or `float` which will be coerced explicitly to `datetime.timedelta.seconds`.
- **tags** (*Union[str, List[str]]*) – List of tags of this link. `[]`
- **copy_records** (*bool*) – Whether to copy records before passing them to outlets. `True`
- **ignore_exceptions** (*bool*) – Whether exceptions in inlets and outlets should be logged and ignored, or raised. `True`
- **inlet_concurrency** (*int*) – How many inlets are allowed to execute concurrently. `9999`
- **immediate_transfer** (*bool*) – Whether governing planners that have `BasePlanner.immediate_transfer` set to `True` should execute this link's transfer once immediately upon starting. `True`
- **processors** (`callable` or list[`callable`]) – *Processors* of this link. `None`
- **groupers** (`callable` or list[`callable`]) – *groupers* of this link. `None`

inlets (*self*) → List[Inlet]
property

Inlets handled by this link.

Returns list[*Inlet*]

add_inlets (*self*, *inlets: Union[Inlet, List[Inlet]]*)
Add inlets to this link. Inlets must be of type Inlet and not currently added to this link.

Parameters **inlets** (*Inlet* or list[*Inlet*]) – inlets to add to this link

Raises *InvalidNodeError* if this link already contains any of the inlets being added.

remove_inlets (*self*, *inlets: Union[Inlet, List[Inlet]]*)
Remove inlets from this link.

Parameters **inlets** (*Inlet* or list[*Inlet*]) – inlets to remove from this link

Raises *InvalidNodeError* if this link doesn't contain any of the inlets being removed.

outlets (*self*) → List[Outlet]
property

Outlets handled by this link.

Returns list[*Outlet*]

add_outlets (*self*, *outlets: Union[Outlet, List[Outlet]]*)
Add outlets to this link. Outlets must be of type Outlet and not currently added to this link.

Parameters **outlets** (*Outlet* or list[*Outlet*]) – outlets to add to this link

Raises *InvalidNodeError* if this link already contains any of the outlets being added.

remove_outlets (*self*, *outlets: Union[Outlet, List[Outlet]]*)
Remove outlets from this link.

Parameters **outlets** (*Outlet* or list[*Outlet*]) – outlets to remove from this link

Raises *InvalidNodeError* if this link doesn't contain any of the outlets being removed.

interval (*self*) → datetime.timedelta
property

Frequency at which this link should transfer.

Returns interval object

Return type `datetime.timedelta`

set_job (*self*, *job*)

Parameters **job** (*Any*) – specify the job this link is executed with.

job (*self*) → Any
property

The job this link is executed with. Job should persist between link transfers. `None`

Returns Job this link is executed with.

name (*self*) → str
property

Deprecated in 0.2.0, will be removed in 1.0. Use *Link.tags* instead.

Name of this Link, equivalent to first tag of this link.

Returns Name of this link

Return type str

tags (*self*) → List[str]
property

The tags of this link. `[]`

Returns Tags of this link

Return type List[str]

transfer (*self*)
Execute one transfer on this link. This will run through all inlets querying them for data, then pass that data to all outlets.

See *Link transfer* to learn more about the transfer.

on_start (*self*)
Called when the governing planner is about to start. Calls *try_start* on all inlets and outlets of this link.

If an inlet or outlet is present in multiple links its on_start will only be called once by whichever link executes first.

on_shutdown (*self*)
Called just after the governing planner has shutdown. Calls *try_shutdown* on all inlets and outlets of this link.

If an inlet or outlet is present in multiple links its on_shutdown will only be called once by whichever link executes first.

2.5.10 databay.outlet

See also:

- *Extending Outlets* to learn how to extend this class correctly.
- `Inlet` representing the corresponding input of the data stream.

class databay.outlet.**Outlet**(*processors: Union[callable, List[callable]] = None*)
databay.outlet.Outlet

Abstract class representing an output of the data stream.

Parameters **processors** (callable or list[callable]) – *Processors* of this outlet. None

Bases: abc.ABC

push(*self*, *records: List[Record]*, *update: da.Update*)
abstractmethod

Push received data.

Override this method to define how this outlet will handle received data.

Parameters

- **records** (list[*Record*]) – List of records generated by inlets. Each top-level element of this array corresponds to one inlet that successfully returned data. Note that inlets could return arrays too, making this a nested array.
- **update** (*Update*) – Update object representing the particular Link transfer.

try_start(*self*)
Wrapper around on_start call that will ensure it only gets executed once.

on_start(*self*)
Called once per outlet just before the governing planner is about to start.

Override this method to provide starting functionality on this outlet.

try_shutdown(*self*)
Wrapper around on_shutdown call that will ensure it only gets executed once.

on_shutdown(*self*)
Called once per outlet just after the governing planner has shutdown.

Override this method to provide shutdown functionality on this outlet.

active(*self*)
property

Whether this outlet is active and ready to push. This variable is set automatically to True on start and to False on shutdown. False

Return type bool

2.5.11 databay.record

class databay.record.**Record**(*payload*, *metadata: dict = None*)
databay.record.Record

Data structure representing the data passed between inlets and outlets.

> **Warning:** You should prefer *Inlet.new_record()* function over instantiating this class directly.

Parameters

- **payload** (*Any*) – Data contained by this record.
- **metadata** (*dict*) – Metadata attached to this record None (Set to empty dict if not provided)

payload(*self*) → dict
property

Returns Data stored in this record.

Return type Any

metadata(*self*) → dict
property

Returns Metadata attached to this record.

Return type dict

2.6 Source Code

2.6.1 inlet

2.6.2 base_planner

2.6.3 errors

2.6.4 outlet

2.6.5 link

2.6.6 record

2.6.7 http_inlet

2.6.8 null_inlet

2.6.9 random_int_inlet

2.6.10 file_inlet

2.6.11 buffers

2.6.12 schedule_planner

2.6.13 aps_planner

2.6.14 inlet_tester

2.6.15 null_outlet

2.6.16 mongo_outlet

2.6.17 csv_outlet

2.6.18 print_outlet

2.6.19 file_outlet

PYTHON MODULE INDEX

d

INDEX

G

H

I

J

L

M

N

O

P

R

S

T

U