Faust Documentation

Release 1.9.0

Robinhood Markets, Inc.

Jan 09, 2020

CONTENTS

```
# Python Streams 🙌 🚀 ❤
# Forever scalable event processing & in-memory durable K/V storage
# w/ asyncio & static typing.
import faust
```

Faust is a stream processing library, porting the ideas from Kafka Streams to Python.

It is used at Robinhood to build high performance distributed systems and real-time data pipelines that process billions of events every day.

Faust provides both *stream processing* and *event processing*, sharing similarity with tools such as Kafka Streams, Apache Spark/Storm/Samza/Flink,

It does not use a DSL, it's just Python! This means you can use all your favorite Python libraries when stream processing: NumPy, PyTorch, Pandas, NLTK, Django, Flask, SQLAlchemy, ++

Faust requires Python 3.6 or later for the new async/await syntax, and variable type annotations.

Here's an example processing a stream of incoming orders:

```
app = faust.App('myapp', broker='kafka://localhost')

# Models describe how messages are serialized:
# {"account_id": "3fae-...", amount": 3}
class Order(faust.Record):
    account_id: str
    amount: int

@app.agent(value_type=Order)
async def order(orders):
    async for order in orders:
        # process infinite stream of orders.
        print(f'Order for {order.account_id}: {order.amount}')
```

The Agent decorator defines a "stream processor" that essentially consumes from a Kafka topic and does something for every event it receives.

The agent is an `async def` function, so can also perform other operations asynchronously, such as web requests.

This system can persist state, acting like a database. Tables are named distributed key/value stores you can use as regular Python dictionaries.

Tables are stored locally on each machine using a super fast embedded database written in C++, called RocksDB.

Tables can also store aggregate counts that are optionally "windowed" so you can keep track of "number of clicks from the last day," or "number of clicks in the last hour." for example. Like Kafka Streams, we support tumbling, hopping and sliding windows of time, and old windows can be expired to stop data from filling up.

For reliability we use a Kafka topic as "write-ahead-log". Whenever a key is changed we publish to the changelog. Standby nodes consume from this changelog to keep an exact replica of the data and enables instant recovery should any of the nodes fail.

To the user a table is just a dictionary, but data is persisted between restarts and replicated across nodes so on failover other nodes can take over automatically.

You can count page views by URL:

```
# Data sent to 'clicks' topic sharded by URL key.
# e.g. key="http://example.com" value="1"
click_topic = app.topic('clicks', key_type=str, value_type=int)
```

(continues on next page)

(continued from previous page)

```
# default value for missing URL will be 0 with `default=int`
counts = app.Table('click_counts', default=int)

@app.agent(click_topic)
async def count_click(clicks):
    async for url, count in clicks.items():
        counts[url] += count
```

The data sent to the Kafka topic is partitioned, which means the clicks will be sharded by URL in such a way that every count for the same URL will be delivered to the same Faust worker instance.

Faust supports any type of stream data: bytes, Unicode and serialized structures, but also comes with "Models" that use modern Python syntax to describe how keys and values in streams are serialized:

```
# Order is a json serialized dictionary,
# having these fields:

class Order(faust.Record):
    account_id: str
    product_id: str
    price: float
    quantity: float = 1.0

orders_topic = app.topic('orders', key_type=str, value_type=Order)

@app.agent(orders_topic)
async def process_order(orders):
    async for order in orders:
        # process each order using regular Python
        total_price = order.price * order.quantity
        await send_order_received_email(order.account_id, order)
```

Faust is statically typed, using the mypy type checker, so you can take advantage of static types when writing applications.

The Faust source code is small, well organized, and serves as a good resource for learning the implementation of Kafka Streams.

Learn more about Faust in the *Introducing Faust* introduction page to read more about Faust, system requirements, installation instructions, community resources, and more.

or go directly to the *Quick Start* tutorial to see Faust in action by programming a streaming application.

then explore the *User Guide* for in-depth information organized by topic.

CHAPTER

ONE

CONTENTS

1.1 Copyright

Faust User Manual

Note: While the Faust *documentation* is offered under the Creative Commons *Attribution-ShareAlike 4.0 International* license the Faust *software* is offered under the BSD License (3 Clause)

1.2 Introducing Faust

Version 1.9.0

Web http://faust.readthedocs.io/

Download http://pypi.org/project/faust

Source http://github.com/robinhood/faust

Keywords distributed, stream, async, processing, data, queue, state management

Table of Contents

1.2.1 What can it do?

Agents Process infinite streams in a straightforward manner using asynchronous generators. The concept of "agents" comes from the actor model, and means the stream processor can execute concurrently on many CPU cores, and on hundreds of machines at the same time.

Use regular Python syntax to process streams and reuse your favorite libraries:

```
@app.agent()
async def process(stream):
    async for value in stream:
        process(value)
```

Tables Tables are sharded dictionaries that enable stream processors to be stateful with persistent and durable data.

Streams are partitioned to keep relevant data close, and can be easily repartitioned to achieve the topology you need.

In this example we repartition an order stream by account id, to count orders in a distributed table:

```
import faust

# this model describes how message values are serialized
# in the Kafka "orders" topic.
class Order(faust.Record, serializer='json'):
    account_id: str
    product_id: str
    amount: int
    price: float

app = faust.App('hello-app', broker='kafka://localhost')
orders_kafka_topic = app.topic('orders', value_type=Order)

# our table is sharded amongst workers cluster-wide, and replicated
# with standby copies to take over if one of the nodes fail.
order_count_by_account = app.Table('order_count', default=int)

@app.agent(orders_kafka_topic)
async def process(orders: faust.Stream[Order]) -> None:
    async for order in orders.group_by(Order.account_id):
        order_count_by_account[order.account_id] += 1
```

If we start multiple instances of this Faust application on many machines, any order with the same account id will be received by the same stream processing agent, so the count updates correctly in the table.

Sharding/partitioning is an essential part of stateful stream processing applications, so take this into account when designing your system, but note that streams can also be processed in round-robin order so you can use Faust for event processing and as a task queue also.

Asynchronous with `asyncio` Faust takes full advantage of `asyncio` and the new `async`/`await` keywords in Python 3.6+ to run multiple stream processors in the same process, along with web servers and other network services.

Thanks to Faust and `asyncio` you can now embed your stream processing topology into your existing `asyncio`/eventlet/Twisted/Tornado applications.

Faust is…

Simple Faust is extremely easy to use. To get started using other stream processing solutions you have complicated hello-world projects, and infrastructure requirements. Faust only requires Kafka, the rest is just Python, so If you know Python you can already use Faust to do stream processing, and it can integrate with just about anything.

Here's one of the easier applications you can make:

```
import faust

class Greeting(faust.Record):
    from_name: str
    to_name: str

app = faust.App('hello-app', broker='kafka://localhost')
topic = app.topic('hello-topic', value_type=Greeting)

@app.agent(topic)
async def hello(greetings):
    async for greeting in greetings:
        print(f'Hello from {greeting.from_name} to {greeting.to_name}')

@app.timer(interval=1.0)
async def example_sender(app):
    await hello.send(
        value=Greeting(from_name='Faust', to_name='you'),
    )

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

You're probably a bit intimidated by the *async* and *await* keywords, but you don't have to know how `asyncio` works to use Faust: just mimic the examples, and you'll be fine.

The example application starts two tasks: one is processing a stream, the other is a background thread sending events to that stream. In a real-life application, your system will publish events to Kafka topics that your processors can consume from, and the background thread is only needed to feed data into our example.

Highly Available Faust is highly available and can survive network problems and server crashes. In the case of node failure, it can automatically recover, and tables have standby nodes that will take over.

Distributed Start more instances of your application as needed.

Fast A single-core Faust worker instance can already process tens of thousands of events every second, and we are reasonably confident that throughput will increase once we can support a more optimized Kafka client.

Flexible Faust is just Python, and a stream is an infinite asynchronous iterator. If you know how to use Python, you already know how to use Faust, and it works with your favorite Python libraries like Django, Flask, SQLAlchemy, NTLK, NumPy, SciPy, TensorFlow, etc.

Faust is used for…

- **Event Processing**
- **Distributed Joins & Aggregations**

- **Machine Learning**
- **Asynchronous Tasks**
- **Distributed Computing**
- **Data Denormalization**
- **Intrusion Detection**
- **Realtime Web & Web Sockets.**
- **and much more…**

1.2.2 How do I use it?

Step 1: Add events to your system

- Was an account created? Publish to Kafka.
- Did a user change their password? Publish to Kafka.
- Did someone make an order, create a comment, tag something, …? Publish it all to Kafka!

Step 2: Use Faust to process those events

Some ideas based on the events mentioned above:

- Send email when an order is dispatches.
- Find orders created with no corresponding dispatch event for more than three consecutive days.
- Find accounts that changed their password from a suspicious IP address.
- Starting to get the idea?

1.2.3 What do I need?

Version Requirements

Faust version 1.0 runs on

Core

- Python 3.6 or later.
- Kafka 0.10.1 or later.

Extensions

- RocksDB 5.0 or later, python-rocksdb

Faust requires Python 3.6 or later, and a running Kafka broker.

There's no plan to support earlier Python versions. Please get in touch if this is something you want to work on.

1.2.4 Extensions

Name	Version	Bundle
rocksdb	5.0	pip install faust[rocksdb]
redis	aredis 1.1	pip install faust[redis]
datadog	0.20.0	pip install faust[datadog]
statsd	3.2.1	pip install faust[statsd]
uvloop	0.8.1	pip install faust[uvloop]
eventlet	1.16.0	pip install faust[eventlet]
yaml	5.1.0	pip install faust[yaml]

Optimizations

These can be all installed using `pip install faust[fast]`:

Name	Version	Bundle
aiodns	1.1.0	pip install faust[aiodns]
cchardet	1.1.0	pip install faust[cchardet]
ciso8601	2.1.0	pip install faust[ciso8601]
cython	0.9.26	pip install faust[cython]
orjson	2.0.0	pip install faust[orjson]
setproctitle	1.1.0	pip install faust[setproctitle]

Debugging extras

These can be all installed using `pip install faust[debug]`:

Name	Version	Bundle
aiomonitor	0.3	pip install faust[aiomonitor]
setproctitle	1.1.0	pip install faust[setproctitle]

Note: See bundles in the *Installation* instructions section of this document for a list of supported setuptools extensions.

To specify multiple extensions at the same time

separate extensions with the comma:

```
$ pip install faust[uvloop,fast,rocksdb,datadog,redis]
```

RocksDB On MacOS Sierra

To install python-rocksdb on MacOS Sierra you need to specify some additional compiler flags:

```
$ CFLAGS='-std=c++11 -stdlib=libc++ -mmacosx-version-min=10.10' \
    pip install -U --no-cache python-rocksdb
```

1.2.5 Design considerations

Modern Python Faust uses current Python 3 features such as `async`/`await` and type annotations. It's statically typed and verified by the mypy type checker. You can take advantage of type annotations when writing Faust applications, but this is not mandatory.

Library Faust is designed to be used as a library, and embeds into any existing Python program, while also including helpers that make it easy to deploy applications without boilerplate.

Supervised The Faust worker is built up by many different services that start and stop in a certain order. These services can be managed by supervisors, but if encountering an irrecoverable error such as not recovering from a lost Kafka connections, Faust is designed to crash.

For this reason Faust is designed to run inside a process supervisor tool such as supervisord, Circus, or one provided by your Operating System.

Extensible Faust abstracts away storages, serializers, and even message transports, to make it easy for developers to extend Faust with new capabilities, and integrate into your existing systems.

Lean The source code is short and readable and serves as a good starting point for anyone who wants to learn how Kafka stream processing systems work.

1.2.6 Getting Help

Mailing list

For discussions about the usage, development, and future of Faust, please join the faust-users mailing list.

Slack

Come chat with us on Slack:

https://join.slack.com/t/fauststream/shared_invite/enQtNDEzMTIyMTUyNzU2LTRkM2Q2ODkwZTk5MzczNmUxOGU0NWYxNzA2

1.2.7 Resources

Bug tracker

If you have any suggestions, bug reports, or annoyances please report them to our issue tracker at https://github.com/robinhood/faust/issues/

1.2.8 License

This software is licensed under the *New BSD License*. See the `LICENSE` file in the top distribution directory for the full license text.

1.3 Playbooks

Release 1.9

Date Jan 09, 2020

1.3.1 Quick Start

Hello World

Application

The first thing you need to get up and running with Faust is to define an application.

The application (or app for short) configures your project and implements common functionality. We also define a topic description, and an agent to process messages in that topic.

Lets create the file *hello_world.py*:

```
import faust

app = faust.App(
    'hello-world',
    broker='kafka://localhost:9092',
    value_serializer='raw',
)

greetings_topic = app.topic('greetings')

@app.agent(greetings_topic)
async def greet(greetings):
    async for greeting in greetings:
        print(greeting)
```

In this tutorial, we keep everything in a single module, but for larger projects, you can create a dedicated package with a submodule layout.

The first argument passed to the app is the `id` of the application, needed for internal bookkeeping and to distribute work among worker instances.

By default Faust will use JSON serialization, so we specify `value_serializer` here as `raw` to avoid deserializing incoming greetings. For real applications you should define models (see *Models, Serialization, and Codecs*).

Here you defined a Kafka topic `greetings` and then iterated over the messages in the topic and printed each one of them.

Note: The application *id* setting (i.e. `'hello-world'` in the example above), should be unique per Faust app in your Kafka cluster.

Starting Kafka

Before running your app, you need to start Zookeeper and Kafka.

Start Zookeeper first:

```
$ $KAFKA_HOME/bin/zookeeper-server-start $KAFKA_HOME/etc/kafka/zookeeper.properties
```

Then start Kafka:

```
$ $KAFKA_HOME/bin/kafka-server-start $KAFKA_HOME/etc/kafka/server.properties
```

Running the Faust worker

Now that you have created a simple Faust application and have Kafka and Zookeeper running, you need to run a worker instance for the application.

Start a worker:

```
$ faust -A hello_world worker -l info
```

Multiple instances of a Faust worker can be started independently to distribute stream processing across machines and CPU cores.

In production, you'll want to run the worker in the background as a daemon. Use the tools provided by your platform, or use something like supervisord.

Use `--help` to get a complete listing of available command-line options:

```
$ faust worker --help
```

Seeing things in Action

At this point, you have an application running, but not much is happening. You need to feed data into the Kafka topic to see Faust print the greetings as it processes the stream, and right now that topic is probably empty.

Let's use the **`faust send`** command to push some messages into the `greetings` topic:

```
$ faust -A hello_world send @greet "Hello Faust"
```

The above command sends a message to the `greet` agent by using the `@` prefix. If you don't use the prefix, it will be treated as the name of a topic:

```
$ faust -A hello_world send greetings "Hello Kafka topic"
```

After sending the messages, you can see your worker start processing them and print the greetings to the console.

Where to go from here...

Now that you have seen a simple Faust application in action, you should dive into the other sections of the *User Guide* or jump right into the *Playbooks* for tutorials and solutions to common patterns.

1.3.2 Tutorial: Count page views

In the *Quick Start* tutorial, we went over a simple example reading through a stream of greetings and printing them to the console. In this playbook we do something more meaningful with an incoming stream, we'll maintain real-time counts of page views from a stream of page views.

Application

As we did in the *Quick Start* tutorial, we first define our application. Let's create the module `page_views.py`:

```
import faust

app = faust.App(
    'page_views',
    broker='kafka://localhost:9092',
    topic_partitions=4,
)
```

The `topic_partitions` setting defines the maximum number of workers we can distribute the workload to (also sometimes referred as the "sharding factor"). In this example, we set this to 4, but in a production app, we ideally use a higher number.

Page View

Let's now define a *model* that each page view event from the stream deserializes into. The record is used for JSON dictionaries and describes fields much like the new dataclasses in Python 3.7:

Create a model for our page view event:

```
class PageView(faust.Record):
    id: str
    user: str
```

Type annotations are used not only for defining static types, but also to define how fields are deserialized, and lets you specify models that contains other models, and so on. See the *Models, Serialization, and Codecs* guide for more information.

Input Stream

Next we define the source topic to read the "page view" events from, and we specify that every value in this topic is of the PageView type.

```
page_view_topic = app.topic('page_views', value_type=PageView)
```

Counts Table

Then we define a *Table*. This is like a Python dictionary, but is distributed across the cluster, partitioned by the dictionary key.

```
page_views = app.Table('page_views', default=int)
```

Count Page Views

Now that we have defined our input stream, as well as a table to maintain counts, we define an agent reading each page view event coming into the stream, always incrementing the count for that page in the table.

Create the agent:

```
@app.agent(page_view_topic)
async def count_page_views(views):
    async for view in views.group_by(PageView.id):
        page_views[view.id] += 1
```

Note: Here we use *group_by* to repartition the input stream by the page id. This is so that we maintain counts on each instance sharded by the page id. This way in the case of failure, when we move the processing of some partition to another node, the counts for that partition (hence, those page ids) also move together.

Now that we written our project, let's try running it to see the counts update in the changelog topic for the table.

Starting Kafka

Before starting a worker, you need to start Zookeeper and Kafka.

First start Zookeeper:

```
$ $KAFKA_HOME/bin/zookeeper-server-start $KAFKA_HOME/etc/kafka/zookeeper.properties
```

Then start Kafka:

```
$ $KAFKA_HOME/bin/kafka-server-start $KAFKA_HOME/etc/kafka/server.properties
```

Starting the Faust worker

Start the worker, similar to what we did in the *Quick Start* tutorial:

```
$ faust -A page_views worker -l info
```

Seeing it in action

Now let's produce some fake page views to see things in action. Send this data to the `page_views` topic:

```
$ faust -A page_views send page_views '{"id": "foo", "user": "bar"}'
```

Look at the changelog topic to see the counts. To look at the changelog topic we will use the Kafka console consumer.

```
$ $KAFKA_HOME/bin/kafka-console-consumer --topic page_views-page_views-changelog --
↪bootstrap-server localhost:9092 --property print.key=True --from-beginning
```

Note: By default the changelog topic for a given `Table` has the format `<app_id>-<table_name>-changelog`

1.3.3 Tutorial: Leader Election

Faust processes streams of data forming pipelines. Sometimes steps in the pipeline require synchronization, but instead of using mutexes, a better solution is to have one of the workers elected as the leader.

An example of such an application is a news crawler. We can elect one of the workers to be the leader, and the leader maintains all subscriptions (the sources to crawl), then periodically tells the other workers in the cluster to process them.

To demonstrate this we implement a straightforward example where we elect one of our workers as the leader. This leader then periodically send out random greetings to be printed out by available workers.

Application

As we did in the *Tutorial: Count page views* tutorial, we first define your application.

Create a module named `leader.py`:

```
# examples/leader.py

import faust

app = faust.App(
    'leader-example',
    broker='kafka://localhost:9092',
    value_serializer='raw',
)
```

Greetings Agent

Next we define the `say` *agent* that will get greetings from the leader and print them out to the console.

Create the agent:

```
@app.agent()
async def say(greetings):
    async for greeting in greetings:
        print(greeting)
```

See also:

- The *Agents - Self-organizing Stream Processors* guide – for more information about agents.

Leader Timer

Now define a *timer* with the `on_leader` flag enabled so it only executes on the leader.

The `timer` will periodically send out a random greeting, to be printed by one of the workers in the cluster.

Create the leader timer:

```
import random

@app.timer(2.0, on_leader=True)
async def publish_greetings():
    print('PUBLISHING ON LEADER!')
    greeting = str(random.random())
    await say.send(value=greeting)
```

Note: The greeting could be picked up by the agent `say` on any one of the running instances.

Starting Kafka

To run the project you first need to start Zookeeper and Kafka.

Start Zookeeper:

```
$ $KAFKA_HOME/bin/zookeeper-server-start $KAFKA_HOME/etc/kafka/zookeeper.properties
```

Then start Kafka:

```
$ $KAFKA_HOME/bin/kafka-server-start $KAFKA_HOME/etc/kafka/server.properties
```

Starting the Faust worker

Start the Faust worker, similarly to how we do it in the *Quick Start* tutorial:

```
$ faust -A leader worker -l info --web-port 6066
```

Let's start two more workers in different terminals on the same machine:

```
$ faust -A leader worker -l info --web-port 6067
```

```
$ faust -A leader worker -l info --web-port 6068
```

Seeing things in Action

Next try to arbitrary shut down (`Control-c`) some of the workers, to see how the leader stays at just *one* worker - electing a new leader upon killing a leader – and to see the greetings printed by the workers.

1.3.4 Overview: Faust vs Kafka Streams

KStream

- `.filter()`
- `.filterNot()`

  Just use the *if* statement:

  ```
  @app.agent(topic)
  async def process(stream):
      async for event in stream:
          if event.amount >= 300.0:
              yield event
  ```

- `.map()`

  Just call the function you want from within the `async for` iteration:

```
@app.agent(Topic)
async def process(stream):
    async for key, event in stream.items():
        yield myfun(key, event)
```

- `.forEach()`

  In KS `forEach` is the same as `map`, but ends the processing chain.

- `.peek()`

  In KS `peek` is the same as `map`, but documents that the action may have a side effect.

- `.mapValues()`:

```
@app.agent(topic)
async def process(stream):
    async for event in stream:
        yield myfun(event)
```

- `.print()`:

```
@app.agent(topic)
async def process(stream):
    async for event in stream:
        print(event)
```

- `.writeAsText()`:

```
@app.agent(topic)
async def process(stream):
    async for key, event in stream.items():
        with open(path, 'a') as f:
            f.write(repr(key, event))
```

- `.flatMap()`
- `.flatMapValues()`

```
@app.agent(topic)
async def process(stream):
    async for event in stream:
        # split sentences into words
        for word in event.text.split():
            yield event.derive(text=word)
```

- `.branch()`

  This is a special case of *filter* in KS, in Faust just write code and forward events as appropriate:

```
app = faust.App('transfer-demo')

# source topic
source_topic = app.topic('transfers')

# destination topics
tiny_transfers = app.topic('tiny_transfers')
small_transfers = app.topic('small_transfers')
large_transfers = app.topic('large_transfers')
```

(continues on next page)

(continued from previous page)

```
@app.agent(source_topic)
async def process(stream):
    async for event in stream:
        if event.amount >= 1000.0:
            event.forward(large_transfers)
        elif event.amount >= 100.0:
            event.forward(small_transfers)
        else:
            event.forward(tiny_transfers)
```

- `.through()`:

```
@app.agent(topic)
async def process(stream):
    async for event in stream.through('other-topic'):
        yield event
```

- `.to()`:

```
app = faust.App('to-demo')
source_topic = app.topic('source')
other_topic = app.topic('other')

@app.agent(source_topic)
async def process(stream):
    async for event in stream:
        event.forward(other_topic)
```

- `.selectKey()`

  Just transform the key yourself:

```
@app.agent(source_topic)
async def process(stream):
    async for key, value in stream.items():
        key = format_key(key)
```

  If you want to transform the key for processors to use, then you have to change the current context to have the new key:

```
@app.agent(source_topic)
async def process(stream):
    async for event in stream:
        event.req.key = format_key(event.req.key)
```

- `groupBy()`

```
@app.agent(source_topic)
async def process(stream):
    async for event in stream.group_by(Withdrawal.account):
        yield event
```

- `groupByKey()`

  ???

- `.transform()`

- `.transformValues()`

  ???

- `.process()`

  Process in KS calls a Processor and is usually used to also call periodic actions (punctuation). In Faust you'd rather create a background task:

```
import faust

# Useless example collecting transfer events
# and summing them up after one second.

class Transfer(faust.Record, serializer='json'):
    amount: float

app = faust.App('transfer-demo')
transfer_topic = app.topic('transfers', value_type=Transfer)

class TransferBuffer:

    def __init__(self):
        self.pending = []
        self.total = 0

    def flush(self):
        for amount in self.pending:
            self.total += amount
        self.pending.clear()
        print('TOTAL NOW: %r' % (total,))

    def add(self, amount):
        self.pending.append(amount)
buffer = TransferBuffer()

@app.agent(transfer_topic)
async def task(transfers):
    async for transfer in transfers:
        buffer.add(transfer.amount)

@app.timer(interval=1.0)
async def flush_buffer():
    buffer.flush()

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

- `join()`
- `outerJoin()`
- `leftJoin()`

  NOT IMPLEMENTED

```
async for event in (s1 & s2).join()
async for event in (s1 & s2).outer_join()
async for event in (s1 & s2).left_join()
```

1.3.5 Overview: Faust vs. Celery

Faust is a stream processor, so what does it have in common with Celery?

If you've used tools such as Celery in the past, you can think of Faust as being able to, not only run tasks, but for tasks to keep history of everything that has happened so far. That is tasks ("agents" in Faust) can keep state, and also replicate that state to a cluster of Faust worker instances.

If you have used Celery you probably know tasks such as this:

```
from celery import Celery

app = Celery(broker='amqp://')

@app.task()
def add(x, y):
    return x + y

if __name__ == '__main__':
    add.delay(2, 2)
```

Faust uses Kafka as a broker, not RabbitMQ, and Kafka behaves differently from the queues you may know from brokers using AMQP/Redis/Amazon SQS/and so on.

Kafka doesn't have queues, instead it has "topics" that can work pretty much the same way as queues. A topic is a log structure so you can go forwards and backwards in time to retrieve the history of messages sent.

The Celery task above can be rewritten in Faust like this:

```
import faust

app = faust.App('myapp', broker='kafka://')

class AddOperation(faust.Record):
    x: int
    y: int

@app.agent()
async def add(stream):
    async for op in stream:
        yield op.x + op.y

@app.command()
async def produce():
    await add.send(value=AddOperation(2, 2))

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

Faust also support storing state with the task (see *Tables and Windowing*), and it supports leader election which is useful for things such as locks.

Learn more about Faust in the *Introducing Faust* introduction page to read more about Faust, system requirements, installation instructions, community resources, and more.

or go directly to the *Quick Start* tutorial to see Faust in action by programming a streaming application.

then explore the *User Guide* for in-depth information organized by topic.

1.3.6 Cheat Sheet

Process events in a Kafka topic

```
orders_topic = app.topic('orders', value_serializer='json')

@app.agent(orders_topic)
async def process_order(orders):
    async for order in orders:
        print(order['product_id'])
```

Describe stream data using models

```
from datetime import datetime
import faust

class Order(faust.Record, serializer='json', isodates=True):
    id: str
    user_id: str
    product_id: str
    amount: float
    price: float
    date_created: datatime = None
    date_updated: datetime = None

orders_topic = app.topic('orders', value_type=Order)

@app.agent(orders_topic)
async def process_order(orders):
    async for order in orders:
        print(order.product_id)
```

Use async. I/O to perform other actions while processing the stream

```
# [...]
@app.agent(orders_topic)
async def process_order(orders):
    session = aiohttp.ClientSession()
    async for order in orders:
        async with session.get(f'http://e.com/api/{order.id}/') as resp:
            product_info = await request.text()
            await session.post(
                f'http://cache/{order.id}/', data=product_info)
```

Buffer up many events at a time

Here we get up to 100 events within a 30 second window:

```
# [...]
async for orders_batch in orders.take(100, within=30.0):
    print(len(orders))
```

Aggregate information into a table

```
orders_by_country = app.Table('orders_by_country', default=int)

@app.agent(orders_topic)
async def process_order(orders):
    async for order in orders.group_by(order.country_origin):
        country = order.country_origin
        orders_by_country[country] += 1
        print(f'Orders for country {country}: {orders_by_country[country]}')
```

Aggregate information using a window

Count number of orders by country, within the last two days:

```
orders_by_country = app.Table(
    'orders_by_country',
    default=int,
).hopping(timedelta(days=2))

async for order in orders_topic.stream():
    orders_by_country[order.country_origin] += 1
    # values in this table are not considered! access .current

    # for the value related to the time of the current event
    print(orders_by_country[order.country_origin].current())
```

1.4 User Guide

Release 1.9

Date Jan 09, 2020

1.4.1 The App - Define your Faust project

"I am not omniscient, but I know a lot."

– Goethe, *Faust: First part*

What is an Application?

An application is an *instance of the library*, and provides the core API of Faust.

The application can define stream processors (agents), topics, channels, web views, CLI commands and more.

To create one you need to provide a name for the application (the id), a message broker, and a driver to use for table storage (optional)

```
>>> import faust
>>> app = faust.App('example', broker='kafka://', store='rocksdb://')
```

It is safe to…

- Run multiple application instances in the same process:

```
>>> app1 = faust.App('demo1')
>>> app2 = faust.App('demo2')
```

- Share an app between multiple threads (the app is *thread safe*).

Application Parameters

You must provide a name for the app, and also you *will want* to set the `broker` and `store` options to configure the broker URL and a storage driver.

Other than that the rest have sensible defaults so you can safely use Faust without changing them.

Here we set the broker URL to Kafka, and the storage driver to RocksDB:

```
>>> app = faust.App(
...     'myid',
...     broker='kafka://kafka.example.com',
...     store='rocksdb://',
... )
```

`kafka://localhost` is used if you don't configure a broker URL. The first part of the URL (`kafka://`), is called the scheme and specifies the driver that you want to use (it can also be the fully qualified path to a Python class).

The storage driver decides how to keep distributed tables locally, and Faust version 1.0 supports two options:

`rocksdb://`	RocksDB an embedded database (production)
`memory://`	In-memory (development)

Using the `memory://` store is OK when developing your project and testing things out, but for large tables, it can take hours to recover after a restart, so you should never use it in production.

RocksDB recovers tables in seconds or less, is embedded and don't require a server or additional infrastructure. It also stores table data on the file system in such a way that tables can exceed the size of available memory.

See also:

***Configuration Reference*: for a full list of supported configuration** settings – these can be passed as keyword arguments when creating the *`faust.App`*.

Actions

`app.topic()` – Create a topic-description

Use the *`topic()`* method to create a topic description, used to tell stream processors what Kafka topic to read from, and how the keys and values in that topic are serialized:

```
topic = app.topic('name_of_topic')

@app.agent(topic)
async def process(stream):
    async for event in stream:
        ...
```

Topic Arguments

- `key_type`/`value_type`: `ModelArg`

  Use the `key_type` and `value_type` arguments to specify the models used for key and value serialization:

  ```
  class MyValueModel(faust.Record):
      name: str
      value: float

  topic = app.topic(
      'name_of_topic',
      key_type=bytes,
      value_type=MyValueModel,
  )
  ```

  The default `key_type` is `bytes` and treats the key as a binary string. The key can also be specified as a model type (`key_type=MyKeyModel`).

  See also:

  - The *Channels & Topics - Data Sources* guide – for more about topics and channels.
  - The *Models, Serialization, and Codecs* guide – for more about models and serialization.

- `key_serializer`/`value_serializer`: `CodecArg`

  The codec/serializer type used for keys and values in this topic.

  If not specified the default will be taken from the *`key_serializer`* and *`value_serializer`* settings.

  See also:

  - The *Codecs* section in the *Models, Serialization, and Codecs* guide – for more information on available codecs, and also how to make your own custom encoders and decoders.

- `partitions`: `int`

  The number of partitions this topic should have. If not specified the default in the *`topic_partitions`* setting is used.

  Note: if this is an automatically created topic, or an externally managed source topic, then please set this value to `None`.

- `retention`: `Seconds`

  Number of seconds (as `float`/`timedelta`) to keep messages in the topic before they can be expired by the server.

- `compacting`: `bool`

  Set to `True` if this should be a compacting topic. The Kafka broker will then periodically compact the topic, only keeping the most recent value for a key.

- `acks`: `bool`

  Enable automatic acknowledgment for this topic. If you disable this then you are responsible for manually acknowledging each event.

- `internal`: `bool`

  If set to `True` this means we own and are responsible for this topic: we are allowed to create or delete the topic.

- `maxsize`: `int`

  The maximum buffer size used for this channel, with default taken from the *`stream_buffer_maxsize`* setting. When this buffer is exceeded the worker will have to wait for agent/stream consumers to catch up, and if the buffer is frequently full this will negatively affect performance.

  Try tweaking the buffer sizes, but also the *`broker_commit_interval`* setting to make sure it commits more frequently with larger buffer sizes.

`app.channel()` – Create a local channel

Use *`channel()`* to create an in-memory communication channel:

```
import faust

app = faust.App('channel')

class MyModel(faust.Record):
    x: int

channel = app.channel(value_type=MyModel)

@app.agent(channel)
async def process(stream):
    async for event in stream:
        print(f'Received: {event!r}')

@app.timer(1.0)
async def populate():
    await channel.send(MyModel(303))
```

See also:

- The *Channels & Topics - Data Sources* guide – for more about topics and channels.
- The *Models, Serialization, and Codecs* guide – for more about models and serialization.

Channel Arguments

- `key_type`/`value_type`: `ModelArg`

  Use the `key_type` and `value_type` arguments to specify the models used for key and value serialization:

  ```
  class MyValueModel(faust.Record):
      name: str
      value: float

  channel = app.channel(key_type=bytes, value_type=MyValueModel)
  ```

- `key_serializer`/`value_serializer`: `CodecArg`

  The codec/serializer type used for keys and values in this channel.

  If not specified the default will be taken from the *`key_serializer`* and *`value_serializer`* settings.

- `maxsize`: `int`

  This is the maximum number of pending messages in the channel. If this number is exceeded any call to *channel.put(value)* will block until something consumes another message from the channel.

  Defaults to the *`stream_buffer_maxsize`* setting.

`app.Table()` – Define a new table

Use *`Table()`* to define a new distributed dictionary; the only required argument is a unique and identifying name. Here we also set a default value so the table acts as a `defaultdict`:

```
transfer_counts = app.Table(
    'transfer_counts',
    default=int,
    key_type=str,
    value_type=int,
)
```

The default argument is passed in as a callable, and in our example calling `int()` returns the number zero, so whenever a key is missing in the table, it's initialized with a value of zero:

```
>>> table['missing']
0

>>> table['also-missing'] += 1
>>> table['also-missing']
1
```

The table needs to relate every update to an associated source topic event, so you must be iterating over a stream to modify a table. Like in this agent where also `.group_by()` is used to repartition the stream by account id, ensuring every unique account delivers to the same agent instance, and that the count-per-account is recorded accurately:

```
@app.agent(transfers_topic)
async def transfer(transfers):
    async for transfer in transfers.group_by(Transfer.account):
        transfer_counts[transfer.account] += 1
```

The agent modifying the table cannot process the source topic out of order, so only agents with `concurrency=1` are allowed to update tables.

See also:

- The *Tables and Windowing* guide – for more information about tables.

  Learn how to create a "windowed table" where aggregate values are placed into configurable time windows, providing you with answers to questions like "what was the value in the last five minutes", or "what was the value of this count like yesterday".

Table Arguments

- `name`: `str`

  The name of the table. This must be *unique* as two tables with the same in the same application will share changelog topics.

- `help`: `str`

  Brief description of table purpose.

- `default`: `Callable[[], Any]`

  User provided function called to get default value for missing keys.

  Without any default this attempt to access a missing key will raise `KeyError`:

```
>>> table = app.Table('nodefault', default=None)

>>> table['missing']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'missing'
```

  With the default callback set to `int`, the same missing key will now set the key to `0` and return `0`:

```
>>> table = app.Table('hasdefault', default=int)

>>> table['missing']
0
```

- `key_type`/`value_type`: `ModelArg`

  Use the `key_type` and `value_type` arguments to specify the models used for serializing/deserializing keys and values in this table.

```
class MyValueModel(faust.Record):
    name: str
    value: float

table = app.Table(key_type=bytes, value_type=MyValueModel)
```

- `store`: `str` or `URL`

  The name of a storage backend to use, or the URL to one.

  Default is taken from the *`store`* setting.

- `partitions`: `int`

  The number of partitions for the changelog topic used by this table.

  Default is taken from the *`topic_partitions`* setting.

- `changelog_topic`: *`Topic`*

  The changelog topic description to use for this table.

  Only for advanced users who know what they're doing.

- `recovery_buffer_size`: `int`

  How often we flush changelog records during recovery. Default is every 1000 changelog messages.

- `standby_buffer_size`: `int`

How often we flush changelog records during recovery. Default is `None` (always).

- `on_changelog_event`: `Callable[[EventT], Awaitable[None]]`

  A callback called for every changelog event during recovery and while keeping table standbys in sync.

`@app.agent()` – Define a new stream processor

Use the *`agent()`* decorator to define an asynchronous stream processor:

```
# examples/agent.py
import faust

app = faust.App('stream-example')

@app.agent()
async def myagent(stream):
    """Example agent."""
    async for value in stream:
        print(f'MYAGENT RECEIVED -- {value!r}')
        yield value

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

Terminology

- "agent" – A named group of actors processing a stream.
- "actor" – An individual agent instance.

No topic was passed to the agent decorator, so an anonymous topic will be created for it. Use the **`faust agents`** program to list the topics used by each agent:

```
$ python examples/agent.py agents
┌Agents──┬──────────────────────────────────────────┬──────────────┐
│ name   │ topic                                    │ help         │
├────────┼──────────────────────────────────────────┼──────────────┤
│ @myagent │ stream-example-examples.agent.myagent  │ Example agent. │
└────────┴──────────────────────────────────────────┴──────────────┘
```

The autogenerated topic name `stream-example-examples.agent.myagent` is generated from the application *`id`* setting, the application *`version`* setting, and the fully qualified path of the agent (`examples.agent.myagent`).

Start a worker to consume from the topic:

```
$ python examples/agent.py worker -l info
```

Next, in a new console, send the agent a value using the **`faust send`** program. The first argument to send is the name of the topic, and the second argument is the value to send (use `--key=k` to specify key). The name of the topic can also start with the `@` character to name an agent instead.

Use `@agent` to send a value of `"hello"` to the topic of our agent:

```
$ python examples/agent.py send @myagent hello
```

Finally, you should see in the worker console that it received our message:

```
MYAGENT RECEIVED -- b'hello'
```

See also:

- The *Agents - Self-organizing Stream Processors* guide – for more information about agents.
- The *Channels & Topics - Data Sources* guide – for more information about channels and topics.

Agent Arguments

- `name`: `str`

  The name of the agent is automatically taken from the decorated function and the module it is defined in.

  You can also specify the name manually, but note that this should include the module name, e.g.: `name='proj.agents.add'`.

- `channel`: *`Channel`*

  The channel or topic this agent should consume from.

- `concurrency`: `int`

  The number of concurrent actors to start for this agent on every worker instance.

  For example if you have an agent processing RSS feeds, a concurrency of `100` means you can process up to hundred RSS feeds at the same time on every worker instance that you start.

  Adding concurrency to your agent also means it will process events in the topic *out of order*, and should you rewind the stream that order may differ when processing the events a second time.

  Concurrency and tables

  Concurrent agents are **not allowed to modify tables**: an exception is raised if this is attempted.

  They are, however, allowed to read from tables.

- `sink`: *`Iterable[SinkT]`*

  For agents that also yield a value: forward the value to be processed by one or more "sinks".

  A sink can be another agent, a topic, or a callback (async or non-async).

  See also:

  Sinks – for more information on using sinks.

- `on_error`: `Callable[[Agent, BaseException], None]`

  Optional error callback to be called when this agent raises an unexpected exception.

- `supervisor_strategy`: `mode.SupervisorStrategyT`

  A supervisor strategy to decide what happens when the agent raises an exception.

  The default supervisor strategy is `mode.OneForOneSupervisor` – restarting one and one actor as they crash.

  Other built-in supervisor strategies include:

  – `mode.OneForAllSupervisor`

If one agent instance of this type raises an exception we will restart all other agent instances of this type.

– `mode.CrashingSupervisor`

If one agent instance of this type raises an exception we will crash the worker instance.

- `**kwargs`

If the `channel` argument is not specified the agent will use an automatically named topic.

Any additional keyword arguments are considered to be configuration for this topic, with support for the same arguments as *`app.topic()`*.

`@app.task()` – Define a new support task.

Use the *`task()`* decorator to define an asynchronous task to be started with the app:

```
@app.task()
async def mytask():
    print('APP STARTED AND OPERATIONAL')
```

The task will be started when the app starts, by scheduling it as an `asyncio.Task` on the event loop. It will only be started once the app is fully operational, meaning it has started consuming messages from Kafka.

See also:

- The *Tasks* section in the *Tasks, Timers, Cron Jobs, Web Views, and CLI Commands* – for more information about defining tasks.

`@app.timer()` – Define a new periodic task

Use the *`timer()`* decorator to define an asynchronous periodic task that runs every 30.0 seconds:

```
@app.timer(30.0)
async def my_periodic_task():
    print('THIRTY SECONDS PASSED')
```

The timer will start 30 seconds after the worker instance has started and is in an operational state.

See also:

- The *Timers* section in the *Tasks, Timers, Cron Jobs, Web Views, and CLI Commands* guide – for more information about creating timers.

Timer Arguments

- `on_leader`: `bool`

If enabled this timer will only execute on one of the worker instances at a time – that is only on the leader of the cluster.

This can be used as a distributed mutex to execute something on one machine at a time.

`@app.page()` – Define a new Web View

Use the *`page()`* decorator to define a new web view from an async function:

```
# examples/view.py
import faust

app = faust.App('view-example')

@app.page('/path/to/view/')
async def myview(web, request):
    print(f'FOO PARAM: {request.query["foo"]}')

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

Next run a worker instance to start the web server on port 6066 (default):

```
$ python examples/view.py worker -l info
```

Then visit your view in the browser by going to http://localhost:6066/path/to/view/:

```
$ open http://localhost:6066/path/to/view/
```

See also:

- The *Web Views* section in the *Tasks, Timers, Cron Jobs, Web Views, and CLI Commands* guide – to learn more about defining views.

`app.main()` – Start the `faust` command-line program.

To have your script extend the **`faust`** program, you can call `app.main()`:

```
# examples/command.py
import faust

app = faust.App('umbrella-command-example')

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

This will use the arguments in `sys.argv` and will support the same arguments as the **`faust`** umbrella command.

To see a list of available commands, execute your program:

```
$ python examples/command.py
```

To get help for a particular subcommand run:

```
$ python examples/command.py worker --help
```

See also:

- The *`main()`* method in the API reference.

`@app.command()` – Define a new command-line command

Use the *`command()`* decorator to define a new subcommand for the **`faust`** command-line program:

```
# examples/command.py
import faust

app = faust.App('example-subcommand')

@app.command()
async def example():
    """This docstring is used as the command help in --help."""
    print('RUNNING EXAMPLE COMMAND')

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

You can now run your subcommand:

```
$ python examples/command.py example
RUNNING EXAMPLE COMMAND
```

See also:

- The *CLI Commands* section in the *Tasks, Timers, Cron Jobs, Web Views, and CLI Commands* guide – for more information about defining subcommands.

  Including how to specify command-line arguments and parameters to your command.

`@app.service()` – Define a new service

The *`service()`* decorator adds a custom `mode.Service` class as a dependency of the app.

What is a Service?

A service is something that can be started and stopped, and Faust is built out of many such services.

The mode library was extracted out of Faust for being generally useful, and Faust uses this library as a dependency.

Examples of classes that are services in Faust include: the *`App`*, a *`stream`*, an *`agent`*, a *`table`*, the `TableManager`, the `Conductor`, and just about everything that is started and stopped is.

Services can also have background tasks, or execute in an OS thread.

You can *decorate a service class* to have it start with the app:

```
# examples/service.py
import faust
from mode import Service

app = faust.App('service-example')

@app.service
class MyService(Service):

    async def on_start(self):
```

(continues on next page)

(continued from previous page)

```
        print('MYSERVICE IS STARTING')

    async def on_stop(self):
        print('MYSERVICE IS STOPPING')

    @Service.task
    async def _background_task(self):
        while not self.should_stop:
            print('BACKGROUND TASK WAKE UP')
            await self.sleep(1.0)

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

To start the app and see it and action run a worker:

```
python examples/service.py worker -l info
```

You can also add services at runtime in application subclasses:

```
class MyApp(App):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.some_service = self.service(SomeService())
```

Application Signals

You may have experience signals in other frameworks such as Django and Celery.

The main difference between signals in Faust is that they accept positional arguments, and that they also come with asynchronous versions for use with `asyncio`.

Signals are an implementation of the Observer design pattern.

App.on_produce_message

New in version 1.6.

sender *faust.App*

arguments `key, value, partition, timestamp, headers`

synchronous This is a synchronous signal (do not use `async def`).

The `on_produce_message` signal as a synchronous signal called before producing messages.

This can be used to attach custom headers to Kafka messages:

```
from typing import Any, List, Tuple
from faust.types import AppT
from mode.utils.compat import want_bytes

@app.on_produce_message.connect()
def on_produce_attach_trace_headers(
        self,
```

(continues on next page)

(continued from previous page)

```
        sender: AppT,
        key: bytes = None,
        value: bytes = None,
        partition: int = None,
        timestamp: float = None,
        headers: List[Tuple[str, bytes]] = None,
        **kwargs: Any) -> None:
    test = current_test()
    if test is not None:
        # Headers at this point is a list of (key, value) pairs.
        # Note2: values in headers must be :class:`bytes`.
        headers.extend([
            (k, want_bytes(v)) for k, v in test.as_headers().items()
        ])
```

App.on_partitions_revoked

sender *faust.App*

arguments *Set[TP]*

The `on_partitions_revoked` signal is an asynchronous signal called after every Kafka rebalance and provides a single argument which is the set of newly revoked partitions.

Add a callback to be called when partitions are revoked:

```
from typing import Set
from faust.types import AppT, TP

@app.on_partitions_revoked.connect
async def on_partitions_revoked(app: AppT,
                                revoked: Set[TP], **kwargs) -> None:
    print(f'Partitions are being revoked: {revoked}')
```

Using `app` as an instance when connecting here means we will only be called for that particular app instance. If you want to be called for all app instances then you must connect to the signal of the class (`App`):

```
@faust.App.on_partitions_revoked.connect
async def on_partitions_revoked(app: AppT,
                                revoked: Set[TP], **kwargs) -> None:
    ...
```

Signal handlers must always accept `**kwargs`.

Signal handler must always accept `**kwargs` so that they are backwards compatible when new arguments are added.

Similarly new arguments must be added as keyword arguments to be backwards compatible.

App.on_partitions_assigned

sender *faust.App*

arguments *Set[TP]*

The `on_partitions_assigned` signal is an asynchronous signal called after every Kafka rebalance and provides a single argument which is the set of assigned partitions.

Add a callback to be called when partitions are assigned:

```
from typing import Set
from faust.types import AppT, TP

@app.on_partitions_assigned.connect
async def on_partitions_assigned(app: AppT,
                                 assigned: Set[TP], **kwargs) -> None:
    print(f'Partitions are being assigned: {assigned}')
```

App.on_configured

sender *faust.App*

arguments *faust.Settings*

synchronous This is a synchronous signal (do not use `async def`).

Called as the app reads configuration, just before the application configuration is set, but after the configuration is read.

Takes arguments: `(app, conf)`, where conf is the *faust.Settings* object being built and is the instance that `app.conf` will be set to after this signal returns.

Use the `on_configured` signal to configure your app:

```
import os
import faust

app = faust.App('myapp')

@app.on_configured.connect
def configure(app, conf, **kwargs):
    conf.broker = os.environ.get('FAUST_BROKER')
    conf.store = os.environ.get('STORE_URL')
```

App.on_before_configured

sender *faust.App*

arguments *none*

synchronous This is a synchronous signal (do not use `async def`).

Called before the app reads configuration, and before the *App.on_configured* signal is dispatched.

Takes only sender as argument, which is the app being configured:

```
@app.on_before_configured.connect
def before_configuration(app, **kwargs):
    print(f'App {app} is being configured')
```

App.on_after_configured

sender *faust.App*

arguments *none*

synchronous This is a synchronous signal (do not use `async def`).

Called after app is fully configured and ready for use.

Takes only sender as argument, which is the app that was configured:

```
@app.on_after_configured.connect
def after_configuration(app, **kwargs):
    print(f'App {app} has been configured.')
```

App.on_worker_init

sender *faust.App*

arguments *none*

synchronous This is a synchronous signal (do not use `async def`).

Called by the **`faust worker`** program (or when using *app.main()*) to apply worker specific customizations.

Takes only sender as argument, which is the app a worker is being started for:

```
@app.on_worker_init.connect
def on_worker_init(app, **kwargs):
    print(f'Working starting for app {app}')
```

Starting the App

You can start a worker instance for your app from the command-line, or you can start it inline in your Python process. To accommodate the many ways you may want to embed a Faust application, starting the app have several possible entry points:

App entry points:

1) **`faust worker`**

The **`faust worker`** program starts a worker instance for an app from the command-line.

You may turn any self-contained module into the faust program by adding this to the end of the file:

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

For packages you can add a `__main__.py` module or setuptools entry points to `setup.py`.

If you have the module name where an app is defined, you can start a worker for it with the *`faust -A`* option:

```
$ faust -A myproj worker -l info
```

The above will import the app from the `myproj` module using `from myproj import app`. If you need to specify a different attribute you can use a fully qualified path:

```
$ faust -A myproj:faust_app worker -l info
```

2) -> *faust.cli.worker.worker* (CLI interface)

This is the **faust worker** program defined as a Python click command.

It is responsible for:

- Parsing the command-line arguments supported by **faust worker**.
- Printing the banner box (you will not get that with entry point 3 or 4).
- Starting the *faust.Worker* (see next step).

3) -> *faust.Worker*

This is used for starting a worker from Python when you also want to install process signal handlers, etc. It supports the same options as on the **faust worker** command-line, but now they are passed in as keyword arguments to *faust.Worker*.

The Faust worker is a subclass of `mode.Worker`, which makes sense given that Faust is built out of many different mode services starting in a particular order.

The *faust.Worker* entry point is responsible for:

- Changing the directory when the `workdir` argument is set.
- Setting the process title (when setproctitle is installed), for more helpful entry in `ps` listings.
- Setting up `logging`: handlers, formatters and level.
- If *--debug* is enabled:
  - Starting the aiomonitor debugging back door.
  - Starting the blocking detector.
- Setting up `TERM` and `INT` signal handlers.
- Setting up the `USR1` cry handler that logs a traceback.
- Starting the web server.
- Autodiscovery (see `autodiscovery`).
- Starting the *faust.App* (see next step).
- Properly shut down of the event loop on exit.

To start a worker,

1) from synchronous code, use `Worker.execute_from_commandline`:

```
>>> worker = Worker(app)
>>> worker.execute_from_commandline()
```

2) or from an `async def` function call `await worker.start()`:

Warning: You will be responsible for gracefully shutting down the event loop.

```
async def start_worker(worker: Worker) -> None:
    await worker.start()
```

(continues on next page)

(continued from previous page)

```
def manage_loop():
    loop = asyncio.get_event_loop()
    worker = Worker(app, loop=loop)
    try:
        loop.run_until_complete(start_worker(worker))
    finally:
        worker.stop_and_shutdown_loop()
```

Multiple apps

If you want your worker to start multiple apps, you would have to pass them in with the `*services` starargs:

```
worker = Worker(app1, app2, app3, app4)
```

This way the extra apps will be started together with the main app, and the main app of the worker (`worker.app`) will end up being the first positional argument (`app1`).

The problem with starting multiple apps is that each app will start a web server by default.

If you want a web server for every app, you must configure the web port for each:

```
apps = [app1, app2, app3, app4]
for i, app in enumerate(apps):
    app.conf.web_port = 6066 + i

worker = Worker(*apps)
```

---

4) -> *faust.App*

The "worker" only concerns itself with the terminal, process signal handlers, logging, debugging mechanisms, etc., the rest is up to the app.

You can call `await app.start()` directly to get a side-effect free instance that can be embedded in any environment. It won't even emit logs to the console unless you have configured `logging` manually, and it won't set up any `TERM`/`INT` signal handlers, which means `finally` blocks won't execute at shutdown.

Start app directly:

```
async def start_app(app):
    await app.start()
```

This will block until the worker shuts down, so if you want to start other parts of your program, you can start this in the background:

```
def start_in_loop(app):
    loop = asyncio.get_event_loop()
    loop.ensure_future(app.start())
```

If your program is written as a set of Mode services, you can simply add the app as a dependency to your service:

```
class MyService(mode.Service):

    def on_init_dependencies(self):
        return [faust_app]
```

Client-Only Mode

The app can also be started in "client-only" mode, which means the app can be used for sending agent RPC requests and retrieving replies, but not start a full Faust worker:

```
await app.start_client()
```

Projects and Directory Layout

Faust is a library; it does not mandate any specific directory layout and integrates with any existing framework or project conventions.

That said, new projects written from scratch using Faust will want some guidance on how to organize, so we include this as a suggestion in the documentation.

Small/Standalone Projects

You can create a small Faust service with no supporting directories at all, we refer to this as a "standalone module": a module that contains everything it needs to run a full service.

The Faust distribution comes with several standalone examples, such as *examples/word_count.py*.

Medium/Large Projects

Projects need more organization as they grow larger, so we convert the standalone module into a directory layout:

```
+ proj/
    - setup.py
    - MANIFEST.in
    - README.rst
    - setup.cfg

    + proj/
        - __init__.py
        - __main__.py
        - app.py

        + users/
        -   __init__.py
        -   agents.py
        -   commands.py
        -   models.py
        -   views.py

        + orders/
            -   __init__.py
            -   agents.py
            -   models.py
            -   views.py
```

Problem: Autodiscovery

Now we have many *@app.agent*/*@app.timer*'/*@app.command* decorators, and models spread across a nested directory. These have to be imported by the program to be registered and used.

Enter the `autodiscover` setting:

```
# proj/app.py
import faust

app = faust.App(
    'proj',
    version=1,
    autodiscover=True,
    origin='proj'    # imported name for this project (import proj -> "proj")
)

def main() -> None:
    app.main()
```

Using the `autodiscover` and setting it to `True` means it will traverse the directory of the origin module to find agents, timers, tasks, commands and web views, etc.

If you want more careful control you can specify a list of modules to traverse instead:

```
app = faust.App(
    'proj',
    version=1,
    autodiscover=['proj.users', 'proj.orders'],
    origin='proj'
)
```

Autodiscovery when using Django

When using *autodiscover=True* in a Django project, only the apps listed in `INSTALLED_APPS` will be traversed.

See also *Django Projects*.

Problem: Entry Point

The `proj/__main__.py` module can act as the entry point for this project:

```
# proj/__main__.py
from proj.app import app
app.main()
```

After creating this module you can now start a worker by doing:

```
python -m proj worker -l info
```

Now you're probably thinking, "I'm too lazy to type python dash em all the time", but don't worry: take it one step further by using setuptools to install a command-line program for your project.

1) Create a `setup.py` for your project.

   This step is not needed if you already have one.

You can read lots about creating your `setup.py` in the setuptools documentation here: https://setuptools.readthedocs.io/en/latest/setuptools.html#developer-s-guide

A minimum example that will work well enough:

```
#!/usr/bin/env python
from setuptools import find_packages, setup

setup(
    name='proj',
    version='1.0.0',
    description='Use Faust to blah blah blah',
    author='Ola Normann',
    author_email='ola.normann@example.com',
    url='http://proj.example.com',
    platforms=['any'],
    license='Proprietary',
    packages=find_packages(exclude=['tests', 'tests.*']),
    include_package_data=True,
    zip_safe=False,
    install_requires=['faust'],
    python_requires='~=3.6',
)
```

For inspiration you can also look to the *setup.py* files in the faust and mode source code distributions.

2) Add the command as a setuptools entry point.

To your `setup.py` add the following argument:

```
setup(
    ...,
    entry_points={
        'console_scripts': [
            'proj = proj.app:main',
        ],
    },
)
```

This essentially defines that the `proj` program runs *from proj.app import main*

3) Install your package using `setup.py` or **`pip`**.

When developing your project locally you should use `setup.py develop` to use the source code directory as a Python package:

```
$ python setup.py develop
```

You can now run the *proj* command you added to `setup.py` in step two:

```
$ proj worker -l info
```

Why use `develop`? You can use `python setup.py install`, but then you have to run that every time you make modifications to the source files.

Another upside to using `setup.py` is that you can distribute your projects as `pip install`-able packages.

Django Projects

Django has their own conventions for directory layout, but your Django reusable apps will want some way to import your Faust app.

We believe the best place to define the Faust app in a Django project, is in a dedicated reusable app. See the `faustapp` app in the `examples/django` directory in the Faust source code distribution.

Miscellaneous

Why use applications?

For special needs, you can inherit from the *`faust.App`* class, and a subclass will have the ability to change how almost everything works.

Comparing the application to the interface of frameworks like Django, there are clear benefits.

In Django, the global settings module means having multiple configurations are impossible, and with an API organized by modules, you sometimes end up with lots of import statements and keeping track of many modules. Further, you often end up monkey patching to change how something works.

The application keeps the library flexible to changes, and allows for many applications to coexist in the same process space.

Reference

See *`faust.App`* in the API reference for a full list of methods and attributes supported.

1.4.2 Agents - Self-organizing Stream Processors

What is an Agent?

An agent is a distributed system processing the events in a stream.

Every event is a message in the stream and is structured as a key/value pair that can be described using *models* for type safety and straightforward serialization support.

Streams can be divided equally in a round-robin manner, or partitioned by the message key; this decides how the stream divides between available agent instances in the cluster.

Create an agent To create an agent, you need to use the `@app.agent` decorator on an async function taking a stream as the argument. Further, it must iterate over the stream using the `async for` keyword to process the stream:

```
# faustexample.py

import faust

app = faust.App('example', broker='kafka://localhost:9092')


@app.agent()
async def myagent(stream):
    async for event in stream:
        ...  # process event
```

Start a worker for the agent The **`faust worker`** program can be used to start a worker from the same directory as the `faustexample.py` file:

```
$ faust -A faustexample worker -l info
```

Whenever a worker is started or stopped, this will force the cluster to rebalance and divide available partitions between all the workers.

Partitioning

When an agent reads from a topic, the stream is partitioned based on the key of the message. For example, the stream could have keys that are account ids, and values that are high scores, then partitioning will decide that any message with the same account id as key, is always delivered to the same agent instance.

Sometimes you'll have to repartition the stream, to ensure you are receiving the right portion of the data. See *Streams - Infinite Data Structures* for more information on the *`Stream.group_by()`* method.

Round-Robin

If you don't set a key (`key=None`), the messages will be delivered to available workers in round-robin order. This is useful to distribute work evenly between a cluster of workers.

Fault tolerance

If the worker for a partition fails, or is blocked from the network for any reason, there's no need to worry because Kafka will move that partition to a worker that's online.

Faust also takes advantage of "standby tables" and a custom partition manager that prefers to promote any node with a full copy of the data, saving startup time and ensuring availability.

This is an agent that adds numbers (full example):

```
# examples/agent.py
import faust

# The model describes the data sent to our agent,
# We will use a JSON serialized dictionary
# with two integer fields: a, and b.
class Add(faust.Record):
    a: int
    b: int

# Next, we create the Faust application object that
# configures our environment.
app = faust.App('agent-example')

# The Kafka topic used by our agent is named 'adding',
# and we specify that the values in this topic are of the Add model.
# (You can also specify the key_type if your topic uses keys).
topic = app.topic('adding', value_type=Add)

@app.agent(topic)
async def adding(stream):
    async for value in stream:
        # Here we receive Add objects, add a + b,
        yield value.a + value.b
```

Starting a worker will start a single instance of this agent:

```
$ faust -A examples.agent worker -l info
```

To send values to it, open a second console to run this program:

```
# examples/send_to_agent.py
import asyncio
from .agent import Add, adding

async def send_value() -> None:
    print(await adding.ask(Add(a=4, b=4)))

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(send_value())
```

```
$ python examples/send_to_agent.py
```

Define commands with the `@app.command` decorator.

You can also use *CLI Commands* to add actions for your application on the command line. Use the `@app.command` decorator to rewrite the example program above (`examples/agent.py`), like this:

```
@app.command()
async def send_value() -> None:
    print(await adding.ask(Add(a=4, b=4)))
```

After adding this to your `examples/agent.py` module, run your new command using the **`faust`** program:

```
$ faust -A examples.agent send_value
```

You may also specify command line arguments and options:

```
from faust.cli import argument, option

@app.command(
    argument('a', type=int, help='First number to add'),
    argument('b', type=int, help='Second number to add'),
    option('--print/--no-print', help='Enable debug output'),
)
async def send_value(a: int, b: int, print: bool) -> None:
    if print:
        print(f'Sending Add({x}, {y})...')
    print(await adding.ask(Add(a, b)))
```

Then pass those arguments on the command line:

```
$ faust -A examples.agent send_value 4 8 --print
Sending Add(4, 8)...
12
```

The *`Agent.ask()`* method adds additional metadata to the message: the return address (reply-to) and a correlating id (correlation_id).

When the agent sees a message with a return address, it will reply with the result generated from that request.

Static types

Faust is typed using the type annotations available in Python 3.6, and can be checked using the mypy type checker.

Add type hints to your agent function like this:

```
from typing import AsyncIterable
from faust import StreamT

@app.agent(topic)
async def adding(stream: StreamT[Add]) -> AsyncIterable[int]:
    async for value in stream:
        yield value.a + value.b
```

The `StreamT` type used for the agent's stream argument is a subclass of `AsyncIterable` extended with the stream API. You could type this call using `AsyncIterable`, but then mypy would stop you with a typing error should you use stream-specific methods such as `.group_by()`, `through()`, etc.

Defining Agents

The Channel

The `channel` argument to the agent decorator defines the source of events that the agent reads from.

This can be:

- A channel

  Channels are in-memory, and work like a `asyncio.Queue`.

  They also form a basic abstraction useful for integrating with many messaging systems (RabbitMQ, Redis, ZeroMQ, etc.)

- A topic description (as returned by *app.topic()*)

  Describes one or more topics to subscribe to, including a recipe of how to deserialize it:

  ```
  topic = app.topic('topic_name1', 'topic_name2',
                    key_type=Model,
                    value_type=Model,
                    ...)
  ```

  Should the topic description provide multiple topic names, the main topic of the agent will be the first topic in that list (`"topic_name1"`).

  The `key_type` and `value_type` describe how to serialize and deserialize messages in the topic, and you provide it as a model (such as *faust.Record*), a *faust.Codec*, or the name of a serializer.

  If not specified it will use the default serializer defined by the app.

Tip: If you don't specify a topic, the agent will use the agent name as the topic: the name will be the fully qualified name of the agent function (e.g., `examples.agent.adder`).

See also:

- The *Channels & Topics - Data Sources* guide – for more information about topics and channels.

The Stream

The agent decorator expects a function taking a single argument (unary).

The stream passed in as the argument to the agent is an async iterable *Stream* instance, created from the topic/channel provided to the decorator:

```
@app.agent(topic_or_channel)
async def myagent(stream):
    async for item in stream:
        ...
```

Iterating over this stream, using the `async for` keyword will iterate over messages in the topic/channel.

If you need to repartition the stream, you may use the *group_by()* method of the Stream API, like in this example where we repartition by account ID:

```
# examples/groupby.py
import faust

class BankTransfer(faust.Record):
    account_id: str
    amount: float

app = faust.App('groupby')
topic = app.topic('groupby', value_type=BankTransfer)

@app.agent(topic)
async def stream(s):
    async for transfer in s.group_by(BankTransfer.account_id):
        # transfers will now be distributed such that transfers
        # with the same account_id always arrives to the same agent
```

(continues on next page)

(continued from previous page)

```
        # instance
        ...
```

See also:

- The *Streams - Infinite Data Structures* guide – for more information about streams.
- The *Channels & Topics - Data Sources* guide – for more information about topics and channels.

Concurrency

Use the `concurrency` argument to start multiple instances of an agent on every worker instance. Each agent instance (actor) will process items in the stream concurrently (and in no particular order).

> **Warning:** Concurrent instances of an agent will process the stream out-of-order, so you cannot mutate *tables* from within the agent function:
>
> An agent having *concurrency* > *1*, can only read from a table, never write.

Here's an agent example that can safely process the stream out of order.

Our hypothetical backend system publishes a message to the Kafka "news" topic every time a news article is published by an author.

We define an agent that consumes from this topic and for every new article will retrieve the full article over HTTP, then store that in a database:

```
class Article(faust.Record, isodates=True):
    url: str
    date_published: datetime

news_topic = app.topic('news', value_type=Article)

@app.agent(news_topic, concurrency=10)
async def imports_news(articles):
    async for article in articles:
        async with app.http_client.get(article.url) as response:
            await store_article_in_db(response)
```

Sinks

Sinks can be used to perform additional actions after an agent has processed an event in the stream, such as forwarding alerts to a monitoring system, logging to Slack, etc. A sink can be callable, async callable, a topic/channel or another agent.

Function Callback Regular functions take a single argument (the result after processing):

```
def mysink(value):
    print(f'AGENT YIELD: {value!r}')

@app.agent(sink=[mysink])
async def myagent(stream):
    async for value in stream:
        yield process_value(value)
```

Async Function Callback Asynchronous functions also work:

```
async def mysink(value):
    print(f'AGENT YIELD: {value!r}')
    # NOTE: This will force the agent instance that yielded this value
    # to sleep for 1.0 second before continuing on the next event
    # in the stream.
    await asyncio.sleep(1)

@app.agent(sink=[mysink])
async def myagent(stream):
    ...
```

Topic Specifying a topic as the sink means the agent will forward all processed values to that topic:

```
agent_log_topic = app.topic('agent_log')

@app.agent(sink=[agent_log_topic])
async def myagent(stream):
    ...
```

Another Agent Specifying another agent as the sink means the agent will forward all processed values to that other agent:

```
@app.agent()
async def agent_b(stream):
    async for event in stream:
        print(f'AGENT B RECEIVED: {event!r}')

@app.agent(sink=[agent_b])
async def agent_a(stream):
    async for event in stream:
        print(f'AGENT A RECEIVED: {event!r}')
```

When agents raise an error

If an agent raises an exception during processing of an *event* will we mark that event as completed? (*acked*)

Currently the source message will be acked and not processed again, simply because it violates “”exactly-once” semantics”.

It is common to think that we can just retry that event, but it is not as easy as it seems. Let’s analyze our options apart from marking the event as complete.

- Retrying

  The retry would have to stop processing of the topic so that order is maintained: the next offset in the topic can only be processed after the event is retried.

  We can move the event to the “back of the queue”, but that means the topic is now out of order.

- Crashing

  Crashing the instance to require human intervention is a choice, but far from ideal considering how common mistakes in code and unexpected exceptions are. It may be better to log the error and have ops replay and reprocess the stream on notification.

Using Agents

Cast or Ask?

When communicating with an agent, you can ask for the result of the request to be forwarded to another topic: this is the *reply_to* topic.

The `reply_to` topic may be the topic of another agent, a source topic populated by a different system, or it may be a local ephemeral topic collecting replies to the current process.

If you perform a `cast`, you're passively sending something to the agent, and it will not reply back.

Systems perform better when no synchronization is required, so you should try to solve your problems in a streaming manner. If B needs to happen after A, try to have A call B instead (which could be accomplished using `reply_to=B`).

`cast(value, *, key=None, partition=None)` A cast is non-blocking as it will not wait for a reply:

```
await adder.cast(Add(a=2, b=2))
```

The agent will receive the request, but it will not send a reply.

`ask(value, *, key=None, partition=None, reply_to=None, correlation_id=None)` Asking an agent will send a reply back to process that sent the request:

```
value = await adder.ask(Add(a=2, b=2))
assert value == 4
```

`send(key, value, partition, reply_to=None, correlation_id=None)` The `Agent.send` method is the underlying mechanism used by `cast` and `ask`.

Use it to send the reply to another agent:

```
await adder.send(value=Add(a=2, b=2), reply_to=another_agent)
```

Streaming Map/Reduce

These map/reduce operations are shortcuts used to stream lots of values into agents while at the same time gathering the results.

`map` streams results as they come in (out-of-order), and `join` waits until all the steps are complete (back-to-order) and return the results in a list with order preserved:

`map(values: Union[AsyncIterable[V], Iterable[V]])` Map takes an async iterable, or a regular iterable, and returns an async iterator yielding results as they come in:

```
async for reply in agent.map([1, 2, 3, 4, 5, 6, 7, 8]):
    print(f'RECEIVED REPLY: {reply!r}')
```

The iterator will start before all the messages have been sent, and should be efficient even for infinite lists.

As the map executes concurrently, the **replies will not appear in any particular order**.

`kvmap(items: Union[AsyncIterable[Tuple[K, V], Iterable[Tuple[K, V]]])` Same as `map`, but takes an async iterable/iterable of `(key, value)` tuples, where the key in each pair is used as the Kafka message key.

`join(values: Union[AsyncIterable[V], Iterable[V]])` Join works like `map` but will wait until all of the values have been processed and returns them as a list in the original order.

The `await` will continue only after the map sequence is over, and all results are accounted for, so do not attempt to use `join` together with infinite data structures ;-)

```
results = await pow2.join([1, 2, 3, 4, 5, 6, 7, 8])
assert results == [1, 4, 9, 16, 25, 36, 49, 64]
```

`kvjoin(items: Union[AsyncIterable[Tuple[K, V]], Iterable[Tuple[K, V]]])` Same as join, but takes an async iterable/iterable of `(key, value)` tuples, where the key in each pair is used as the message key.

1.4.3 Streams - Infinite Data Structures

> *"Everything transitory is but an image."*
>
> – Goethe, *Faust: Part II*

Basics

A stream is an infinite async iterable, consuming messages from a channel/topic:

```
@app.agent(my_topic)
async def process(stream):
    async for value in stream:
        ...
```

The above *agent* is how you usually define stream processors in Faust, but you can also create stream objects manually at any point with the caveat that this can trigger a Kafka rebalance when doing so at runtime:

```
stream = app.stream(my_topic)  # or: my_topic.stream()
async for value in stream:
    ...
```

The stream *needs to be iterated over* to be processed, it will not be active until you do.

When iterated over the stream gives deserialized values, but you can also iterate over key/value pairs (using `items()`), or raw messages (using `events()`).

Keys and values can be bytes for manual deserialization, or `Model` instances, and this is decided by the topic's `key_type` and `value_type` arguments.

See also:

- The *Channels & Topics - Data Sources* guide – for more information about channels and topics.
- The *Models, Serialization, and Codecs* guide – for more information about models and serialization.

The easiest way to process streams is to use *agents*, but you can also create a stream manually from any topic/channel.

Here we define a model for our stream, create a stream from the "withdrawals" topic and iterate over it:

```
class Withdrawal(faust.Record):
    account: str
    amount: float

async for w in app.topic('withdrawals', value_type=Withdrawal).stream():
    print(w.amount)
```

Do note that the worker must be started first (or at least the app), for this to work, and the stream iterator needs to be started as an `asyncio.Task`, so a more practical example is:

```
import faust

class Withdrawal(faust.Record):
    account: str
    amount: float

app = faust.App('example-app')

withdrawals_topic = app.topic('withdrawals', value_type=Withdrawal)

@app.task
async def mytask():
    async for w in withdrawals_topic.stream():
        print(w.amount)

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

You may also treat the stream as a stream of bytes values:

```
async for value in app.topic('messages').stream():
    # the topic description has no value_type, so values
    # are now the raw message value in bytes.
    print(repr(value))
```

Processors

A stream can have an arbitrary number of processor callbacks that are executed as values go through the stream.

These are normally used in Faust applications, but are useful for libraries that extend the functionality of streams.

A processor takes a value as argument and returns a value:

```
def add_default_language(value: MyModel) -> MyModel:
    if not value.language:
        value.language = 'US'
    return value

async def add_client_info(value: MyModel) -> MyModel:
    value.client = await get_http_client_info(value.account_id)
    return value

s = app.stream(my_topic,
               processors=[add_default_language, add_client_info])
```

Note: Processors can be async callable, or normal callable.

Since the processors are stored in an ordered list, the processors above will execute in order and the final value going out of the stream will be the reduction after all processors are applied:

```
async for value in s:
    # all processors applied here so `value`
    # will be equivalent to doing:
    #   value = add_default_language(add_client_info(value))
```

Message Life Cycle

Kafka Topics

Every Faust worker instance will start a single Kafka consumer responsible for fetching messages from all subscribed topics.

Every message in the topic have an offset number (where the first message has an offset of zero), and we use a single offset to track the messages that consumers do not want to see again.

The Kafka consumer commits the topic offsets every three seconds in a background task. The default interval is defined by the *broker_commit_interval* setting.

As we only have one consumer, and multiple agents can subscribe to the same topic, we need a smart way to track when those events have processed so we can commit and advance the consumer group offset.

We use reference counting for this, so when you define an agent that iterates over the topic as a stream:

```
@app.agent(topic)
async def process(stream):
    async for value in stream:
        print(value)
```

The act of starting that stream iterator will add the topic to the Conductor service. This internal service is responsible for forwarding messages received by the consumer to the streams:

```
[Consumer] -> [Conductor] -> [Topic] -> [Stream]
```

The `async for` is what triggers this, and the agent code above is roughly equivalent to:

```
async def custom_agent(app: App, topic: Topic):
    topic_iterator = aiter(topic)
    app.topics.add(topic)  # app.topics is the Conductor
    stream = Stream(topic_iterator, app=app)
    async for value in stream:
        print(value)
```

If two agents use streams subscribed to the same topic:

```
topic = app.topic('orders')

@app.agent(topic)
async def processA(stream):
    async for value in stream:
        print(f'A: {value}')

@app.agent(topic)
async def processB(stream):
    async for value in stream:
        print(f'B: {value}')
```

The Conductor will forward every message received on the "orders" topic to both of the agents, increasing the reference count whenever it enters an agents stream.

The reference count decreases when the event is *acknowledged*, and when it reaches zero the consumer will consider that offset as "done" and can commit it.

Acknowledgment

The acknowledgment signifies that the event processing is complete and should not happen again.

An event is automatically acknowledged when:

- The agent stream advances to a new event (`Stream.__anext__`)
- An exception occurs in the agent during event processing.
- The application shuts down, or a rebalance is required, and the stream finished processing the event.

What this means is that an event is acknowledged when your agent is finished handling it, but you can also manually control when it happens.

To manually control when the event is acknowledged, and its reference count decreased, use `await event.ack()`

async for event in stream.events(): print(event.value) await event.ack()

You can also use `async for` on the event:

```
async for event in stream.events():
    async with event:
        print(event.value)
        # event acked when exiting this block
```

Note that the conditions in automatic acknowledgment still apply when manually acknowledging a message.

Combining streams

Streams can be combined, so that you receive values from multiple streams in the same iteration:

```
>>> s1 = app.stream(topic1)
>>> s2 = app.stream(topic2)
>>> async for value in (s1 & s2):
...     ...
```

Mostly this is useful when you have two topics having the same value type, but can be used in general.

If you have two streams that you want to process independently you should rather start individual tasks:

```
@app.agent(topic1)
async def process_stream1(stream):
    async for value in stream:
        ...


@app.agent(topic2)
async def process_stream2(stream):
    async for value in stream:
        ...
```

Operations

`group_by()` – Repartition the stream

The *`Stream.group_by()`* method repartitions the stream by taking a "key type" as argument:

```
import faust

class Order(faust.Record):
    account_id: str
    product_id: str
    amount: float
    price: float

app = faust.App('group-by-example')
orders_topic = app.topic('orders', value_type=Order)

@app.agent(orders_topic)
async def process(orders):
    async for order in orders.group_by(Order.account_id):
        ...
```

In the example above the "key type" is a field descriptor, and the stream will be repartitioned by the account_id field found in the deserialized stream value.

The new stream will be using a new intermediate topic where messages have account ids as key, and this is the stream that the agent will finally be iterating over.

Note: `Stream.group_by()` returns a new stream subscribing to the intermediate topic of the group by operation.

Apart from field descriptors, the key type argument can also be specified as a callable, or an async callable, so if you're not using models to describe the data in streams you can manually extract the key used for repartitioning:

```
def get_order_account_id(order):
    return json.loads(order)['account_id']

@app.agent(app.topic('order'))
async def process(orders):
    async for order in orders.group_by(get_order_account_id):
        ...
```

See also:

- The *Models, Serialization, and Codecs* guide – for more information on field descriptors and models.
- The *faust.Stream.group_by()* method in the API reference.

`items()` – Iterate over keys and values

Use *Stream.items()* to get access to both message key and value at the same time:

```
@app.agent()
async def process(stream):
    async for key, value in stream.items():
        ...
```

Note that this changes the type of what you iterate over from `Stream` to `AsyncIterator`, so if you want to repartition the stream or similar, `.items()` need to be the last operation:

```
async for key, value in stream.through('foo').group_by(M.id).items():
    ...
```

`events()` – Access raw messages

Use *Stream.events()* to iterate over raw `Event` values, including access to original message payload and message meta data:

```
@app.agent
async def process(stream):
    async for event in stream.events():
        message = event.message
        topic = event.message.topic
        partition = event.message.partition
        offset = event.message.offset

        key_bytes = event.message.key
        value_bytes = event.message.value

        key_deserialized = event.key
```

(continues on next page)

(continued from previous page)

```
        value_deserialized = event.value

        async with event:  # use `async with event:` for manual ack
            process(event)
            # event will be acked when this block returns.
```

See also:

- The *faust.Event* class in the API reference – for more information about events.
- The *faust.types.tuples.Message* class in the API reference – for more information about the fields available in `event.message`.

`take()` – Buffer up values in the stream

Use *Stream.take()* to gather up multiple events in the stream before processing them, for example to take 100 values at a time:

```
@app.agent()
async def process(stream):
    async for values in stream.take(100):
        assert len(values) == 100
        print(f'RECEIVED 100 VALUES: {values}')
```

The problem with the above code is that it will block forever if there are 99 messages and the last hundredth message is never received.

To solve this add a `within` timeout so that up to 100 values will be processed within 10 seconds:

```
@app.agent()
async def process(stream):
    async for values in stream.take(100, within=10):
        print(f'RECEIVED {len(values)}: {values}')
```

The above code works better: if values are constantly being streamed it will process hundreds and hundreds without delay, but if there are long periods of time with no events received it will still process what it has gathered.

`enumerate()` – Count values

Use *Stream.enumerate()* to keep a count of the number of values seen so far in a stream.

This operation works exactly like the Python `enumerate()` function, but for an asynchronous stream:

```
@app.agent()
async def process(stream):
    async for i, value in stream.enumerate():
        ...
```

The count will start at zero by default, but `enumerate` also accepts an optional starting point argument.

See also:

- The `faust.utils.aiter.aenumerate()` function – for a general version of `enumerate()` that let you enumerate any async iterator, not just streams.
- The `enumerate()` function in the Python standard library.

`through()` – Forward through another topic

Use *`Stream.through()`* to forward every value to a new topic, and replace the stream by subscribing to the new topic:

```
source_topic = app.topic('source-topic')
destination_topic = app.topic('destination-topic')

@app.agent()
async def process(stream):
    async for value in stream.through(destination_topic):
        # we are now iterating over stream(destination_topic)
        print(value)
```

You can also specify the destination topic as a string:

```
# [...]
async for value in stream.through('foo'):
    ...
```

Through is especially useful if you need to convert the number of partitions in a source topic, by using an intermediate table.

If you simply want to forward a value to another topic, you can send it manually, or use the echo recipe below:

```
@app.agent()
async def process(stream):
    async for value in stream:
        await other_topic.send(value)
```

`filter()` – Filter values to omit from stream.

New in version 1.7.

This method is useful for filtering events before repartitioning a stream.

Takes a single argument which must be a callable, either a normal function or an *async def* function.

Example:

```
@app.agent()
async def process(stream):
    async for value in stream.filter(lambda: v > 1000).group_by(...):
        ...
```

`echo()` – Repeat to one or more topics

Use `echo()` to repeat values received from a stream to another channel/topic, or many other channels/topics:

```
@app.agent()
async def process(stream):
    async for event in stream.echo('other_topic'):
        ...
```

The operation takes one or more topics, as string topic names or *`app.topic`*, so this also works:

```
source_topic = app.topic('sourcetopic')
echo_topic1 = app.topic('source-copy-1')
echo_topic2 = app.topic('source-copy-2')

@app.agent(source_topic)
async def process(stream):
    async for event in stream.echo(echo_topic1, echo_topic2):
        ...
```

See also:

- The *Channels & Topics - Data Sources* guide – for more information about channels and topics.

Reference

Note: Do not create `Stream` objects directly, instead use: `app.stream` to instantiate new streams.

1.4.4 Channels & Topics - Data Sources

Basics

Faust agents iterate over streams, and streams iterate over channels.

A channel is a construct used to send and receive messages, then we have the "topic", which is a named-channel backed by a Kafka topic.

Streams read from channels (either a local-channel or a topic).

`Agent` <-> `Stream` <-> `Channel`

Topics are named-channels backed by a transport (to use e.g. Kafka topics):

`Agent` <-> `Stream` <-> `Topic` <-> `Transport` <-> aiokafka

Faust defines these layers of abstraction so that agents can send and receive messages using more than one type of transport.

Topics are highly Kafka specific, while channels are not. That makes channels more natural to subclass should you require a different means of communication, for example using RabbitMQ (AMQP), Stomp, MQTT, NSQ, ZeroMQ, etc.

Channels

A **channel** is a buffer/queue used to send and receive messages. This buffer could exist in-memory in the local process only, or transmit serialized messages over the network.

You can create channels manually and read/write from them:

```
async def main():
    channel = app.channel()

    await channel.put(1)

    async for event in channel:
        print(event.value)
        # the channel is infinite so we break after first event
        break
```

Reference

Sending messages to channel

class faust.**Channel**

async send(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Send message to channel.

Return type Awaitable[*RecordMetadata*]

as_future_message(*key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, eager_partitioning: bool = False*) → faust.types.tuples.FutureMessage

Create promise that message will be transmitted.

Return type *FutureMessage*[]

async publish_message(*fut: faust.types.tuples.FutureMessage, wait: bool = True*) → Awaitable[faust.types.tuples.RecordMetadata]

Publish message to channel.

This is the interface used by `topic.send()`, etc. to actually publish the message on the channel after being buffered up or similar.

It takes a `FutureMessage` object, which contains all the information required to send the message, and acts as a promise that is resolved once the message has been fully transmitted.

Return type Awaitable[*RecordMetadata*]

Declaring

Note: Some channels may require you to declare them on the server side before they're used. Faust will create topics considered internal but will not create or modify "source topics" (i.e., exposed for use by other Kafka applications).

To define a topic as internal use `app.topic('name', ..., internal=True)`.

class faust.**Channel**

maybe_declare()
Declare/create this channel, but only if it doesn't exist.

Return type None

async declare() → None
Declare/create this channel.

This is used to create this channel on a server, if that is required to operate it.

Return type None

Topics

A *topic* is a **named channel**, backed by a Kafka topic. The name is used as the address of the channel, to share it between multiple processes and each process will receive a partition of the topic.

1.4.5 Models, Serialization, and Codecs

Basics

Models describe the fields of data structures used as keys and values in messages. They're defined using a `NamedTuple`-like syntax:

```
class Point(Record, serializer='json'):
    x: int
    y: int
```

Here we define a "Point" record having `x`, and `y` fields of type int.

A record is a model of the dictionary type, having keys and values of a certain type.

When using JSON as the serialization format, the Point model serializes to:

```
>>> Point(x=10, y=100).dumps()
{"x": 10, "y": 100}
```

To temporarily use a different serializer, provide that as an argument to `.dumps`:

```
>>> Point(x=10, y=100).dumps(serializer='pickle')  # pickle + Base64
b'gAN9cQAoWAEAAAB4cQFLClgBAAAAeXECS2RYBwAAAF9fZmF1c3RxA31xBFg
CAAAAbnNxBVgOAAAAX19tYWluX18uUG9pbnRxBnN1Lg=='
```

"Record" is the only type supported, but in the future we also want to have arrays and other data structures.

In use

Models are useful when data needs to be serialized/deserialized, or whenever you just want a quick way to define data.

In Faust we use models to:

- Describe the data used in streams (topic keys and values).
- HTTP requests (POST data).

For example here's a topic where both keys and values are points:

```
my_topic = faust.topic('mytopic', key_type=Point, value_type=Point)

@app.agent(my_topic)
async def task(events):
    async for event in events:
        print(event)
```

> **Warning:** Changing the type of a topic is backward incompatible change. You need to restart all Faust instances using the old key/value types.
>
> The best practice is to provide an upgrade path for old instances.

The topic already knows what type is required, so when sending data you just provide the values as-is:

```
await my_topic.send(key=Point(x=10, y=20), value=Point(x=30, y=10))
```

Anonymous Agents

An "anonymous" agent does not use a topic description.

Instead the agent will automatically create and manage its own topic under the hood.

To define the key and value type of such an agent just pass them as keyword arguments:

```
@app.agent(key_type=Point, value_type=Point)
async def my_agent(events):
    async for event in events:
        print(event)
```

Now instead of having a topic where we can send messages, we can use the agent directly:

```
await my_agent.send(key=Point(x=10, y=20), value=Point(x=30, y=10))
```

Schemas

A "schema" configures both key and value type for a topic, and also the serializers used.

Schemas are also able to read the headers of Kafka messages, and so can be used for more complex serialization support, such as Protocol Buffers or Apache Thrift.

To define a topic using a schema:

```
schema = faust.Schema(
    key_type=Point,
    value_type=Point,
    key_serializer='json',
    value_serializer='json',
)

topic = app.topic('mytopic', schema=schema)
```

If any of the serializer arguments are omitted, the default from the app configuration will be used.

Schemas can also be used with "anonymous agents" (see above)

```
@app.agent(schema=schema)
async def myagent(stream):
    async for value in stream:
        print(value)
```

Schemas are most useful when extending Faust, for example defining a schema that reads message key and value type from Kafka headers:

```
import faust
from faust.types import ModelT
from faust.types.core import merge_headers
from faust.models import registry

class Autodetect(faust.Schema):

    def loads_key(self, app, message, *,
                  loads=None,
                  serializer=None):
        if loads is None:
            loads = app.serializers.loads_key
        # try to get key_type and serializer from Kafka headers
        headers = dict(message.headers)
        key_type_name = headers.get('KeyType')
        serializer = serializer or headers.get('KeySerializer')
        if key_type_name:
            key_type = registry[key_type]
            return loads(key_type, message.key,
                   serializer=serializer)
        else:
            return super().loads_key(
```

(continues on next page)

(continued from previous page)

```
                app, message, loads=loads, serializer=serializer)

    def loads_value(self, app, message, *,
                    loads=None,
                    serializer=None):
        if loads is None:
            loads = app.serializers.loads_value
        # try to get key_type and serializer from Kafka headers
        headers = dict(message.headers)
        value_type_name = headers.get('ValueType')
        serializer = serializer or headers.get('ValueSerializer')
        if value_type_name:
            value_type = registry[value_type]
            return loads(value_type, message.key,
                   serializer=serializer)
        else:
            return super().loads_value(
                app, message, loads=loads, serializer=serializer)

    def on_dumps_key_prepare_headers(self, key, headers):
        # if key is a model, set the KeyType header to the models
        # registered name.

        if isinstance(key, ModelT):
            key_type_name = key._options.namespace
            return merge_headers(headers, {'KeyType': key_type_name})
        return headers

    def on_dumps_value_prepare_headers(self, value, headers):
        if isinstance(value, ModelT):
            value_type_name = value._options.namespace
            return merge_headers(headers, {'ValueType': value_type_name})
        return headers

    app = faust.App('id')
    my_topic = app.topic('mytopic', schema=Autodetect())
```

Manual Serialization

Models are not required to read data from a stream.

To deserialize streams manually, use a topic with bytes values:

```
topic = app.topic('custom', value_type=bytes)

@app.agent
async def processor(stream):
    async for payload in stream:
        data = json.loads(payload)
```

To integrate with external systems, *Codecs* help you support serialization and de-serialization to and from any format. Models describe the form of messages, and codecs explain how they're serialized, compressed, encoded, and so on.

The default codec is configured by the applications `key_serializer` and `value_serializer` arguments:

```
app = faust.App(key_serializer='json')
```

Individual models can override the default by specifying a `serializer` argument when creating the model class:

```
class MyRecord(Record, serializer='json'):
    ...
```

Codecs may also be combined to provide multiple encoding and decoding stages, for example `serializer='json|binary'` will serialize as JSON then use the Base64 encoding to prepare the payload for transmission over textual transports.

See also:

- The *Codecs* section – for more information about codecs and how to define your own.

Sending/receiving raw values

You don't have to use models to deserialize events in topics. instead you may omit the `key_type`/`value_type` options, and instead use the `key_serializer`/`value_serializer` arguments:

```
# examples/nondescript.py
import faust

app = faust.App('values')
transfers_topic = app.topic('transfers')
large_transfers_topic = app.topic('large_transfers')

@app.agent(transfers_topic)
async def find_large_transfers(transfers):
    async for transfer in transfers:
        if transfer['amount'] > 1000.0:
            await large_transfers_topic.send(value=transfer)

async def send_transfer(account_id, amount):
    await transfers_topic.send(value={
        'account_id': account_id,
        'amount': amount,
    })
```

The `raw` serializer will provide you with raw text/bytes (the default is bytes, but use `key_type=str` to specify text):

```
transfers_topic = app.topic('transfers', value_serializer='raw')
```

You may also specify any other supported codec, such as json to use that directly:

```
transfers_topic = app.topic('transfers', value_serializer='json')
```

Model Types

Records

A record is a model based on a dictionary/mapping.

Here's a simple record describing a 2d point, having two required fields:

```
class Point(faust.Record):
    x: int
    y: int
```

To create a new point, provide the fields as keyword arguments:

```
>>> point = Point(x=10, y=20)
>>> point
<Point: x=10, y=20>
```

If you forget to pass a required field, we throw an error:

```
>>> point = Point(x=10)
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/opt/devel/faust/faust/models/record.py", line 96, in __init__
    self._init_fields(fields)
File "/opt/devel/faust/faust/models/record.py", line 106, in _init_fields
    type(self).__name__, ', '.join(sorted(missing))))
TypeError: Point missing required arguments: y
```

If you don't want it to be an error, make it an optional field:

```
class Point(faust.Record, serializer='json'):
    x: int
    y: int = 0
```

You may now omit the `y` field when creating points:

```
>>> point = Point(x=10)
<Point: x=10 y=0>
```

Note: The order is important here: all optional fields must be defined **after** all requred fields.

This is not allowed:

```
class Point(faust.Record, serializer='json'):
    x: int
    y: int = 0
    z: int
```

but this works:

```
class Point(faust.Record, serializer='json')
    x: int
    z: int
    y: int = 0
```

Fields

Records may have fields of arbitrary types and both standard Python types and user defined classes will work.

Note that field types must support serialization, otherwise we cannot reconstruct the object back to original form.

Fields may refer to other models:

```
class Account(faust.Record, serializer='json'):
    id: str
    balance: float

class Transfer(faust.Record, serializer='json'):
    account: Account
    amount: float

transfer = Transfer(
    account=Account(id='RBH1235678', balance=13000.0),
    amount=1000.0,
)
```

The field type is a type annotation, so you can use the mypy type checker to verify arguments passed have the correct type.

We do not perform any type checking at runtime.

Collections

Fields can be collections of another type.

For example a User model may have a list of accounts:

```
from typing import List
import faust


class User(faust.Record):
    accounts: List[Account]
```

Not only lists are supported, you can also use dictionaries, sets and others.

Consult this table of supported annotations:

Collection	Recognized Annotations
List	• `List[ModelT]` • `Sequence[ModelT]` • `MutableSequence[ModelT]`
Set	• `AbstractSet[ModelT]` • `Set[ModelT]` • `MutableSet[ModelT]`
Tuple	• `Tuple[ModelT, ...]` • `Tuple[ModelT, ModelT, str]`
Mapping	• `Dict[KT, ModelT]` • `Dict[ModelT, ModelT]` • `Mapping[KT, ModelT]` • `MutableMapping[KT, ModelT]`

From this table we can tell that we may have a *mapping* of username to account:

```
from typing import Mapping
import faust


class User(faust.Record):
    accounts: Mapping[str, Account]
```

Faust will then automatically reconstruct the `User.accounts` field into a mapping of account-ids to `Account` objects.

Coercion

By default we do not force types, this is for backward compatibility with older Faust application.

This means that a field of type `str` will happily accept `None` as value, and any other type.

If you want strict types enable the `coerce` option:

```
class X(faust.Record, coerce=True):
    foo: str
    bar: Optional[str]
```

Here, the `foo` field will be required to be a string, while the `bar` field can have `None` values.

Tip: Having `validation=True` implies `coerce=True` but will additionally enable field validation.

See *Validation* for more information.

Coercion also enables automatic conversion to and from `datetime` and `Decimal`.

You may also disable coercion for the class, but enable it for individual fields by writing explicit field descriptors:

```
import faust
from faust.models.fields import DatetimeField, StringField

class Account(faust.Record):
    user_id: str = StringField(coerce=True)
    date_joined: datetime = DatetimeField(coerce=False)
    login_dates: List[datetime] = DatetimeField(coerce=True)
```

datetime

When using JSON we automatically convert `datetime` fields into ISO-8601 text format, and automatically convert back into into `datetime` when deserializing.

```
from datetime import datetime
import faust

class Account(faust.Record, coerce=True, serializer='json'):
    date_joined: datetime
```

Other date formats

The default date parser supports ISO-8601 only. To support this format and many other formats (such as `'Sat Jan 12 00:44:36 +0000 2019'`) you can select to use python-dateutil as the parser.

To change the date parsing function for a model globally:

```
from dateutil.parser import parse as parse_date

class Account(faust.Record, coerce=True, date_parser=parse_date):
    date_joined: datetime
```

To change the date parsing function for a specific field:

```
from dateutil.parser import parse as parse_date
from faust.models.fields import DatetimeField

class Account(faust.Record, coerce=True):
    # date_joined: supports ISO-8601 only (default)
    date_joined: datetime

    # date_last_login: comes from weird system with more human
    # readable dates ('Sat Jan 12 00:44:36 +0000 2019'),
    # The dateutil parser can handle many different date and time
    # formats.
    date_last_login: datetime = DatetimeField(date_parser=parse_date)
```

Decimal

JSON doesn't have a high precision decimal field type so if you require high precision you must use `Decimal`.

The built-in JSON encoder will convert these to strings in the json payload, that way we do not lose any precision.

```
from decimal import Decimal
import faust

class Order(faust.Record, coerce=True, serializer='json'):
    price: Decimal
    quantity: Decimal
```

Abstract Models

To create a model base class with common functionality, mark the model class with `abstract=True`.

Abstract models must be inherited from, and cannot be instantiated directly.

Here's an example base class with default fields for creation time and last modified time:

```
class MyBaseRecord(Record, abstract=True):
    time_created: float = None
    time_modified: float = None
```

Inherit from this model to create a new model having the fields by default:

```
class Account(MyBaseRecord):
    id: str

account = Account(id='X', time_created=3124312.3442)
print(account.time_created)
```

Positional Arguments

The best practice when creating model instances is to use keyword arguments, but positional arguments are also supported!

The point `Point(x=10, y=30)` may also be expressed as `Point(10, 30)`.

Back to why this is not a good practice, consider the case of inheritance:

```
import faust

class Point(faust.Record):
    x: int
    y: int

class XYZPoint(Point):
    z: int

point = XYZPoint(10, 20, 30)
assert (point.x, point.y, point.z) == (10, 20, 30)
```

To deduce the order arguments we now have to consider the inheritance tree, this is difficult without looking up the source code.

This quickly turns even more complicated when we add multiple inheritance into the mix:

```
class Point(AModel, BModel):
    ...
```

We suggest using positional arguments only for simple classes such as the Point example, where inheritance of additional fields is not used.

Fields with the same name as a reserved keyword

Sometimes data you want to describe data will contan field names that collide with a reserved Python keyword.

One such example is a field named `in`. You cannot define a model like this:

```
class OpenAPIParameter(Record):
    in: str = 'query'
```

doing so will result in a `NameError` exception being raised.

To properly support this, you need to rename the field but specify an alternative `input_name`:

```
from faust.models.fields import StringField

class OpenAPIParameter(Record):
    location: str = StringField(default='query', input_name='in')
```

The `input_name` here describes the name of the field in serialized payloads. There's also a corresponding `output_name` that can be used to specify what field name this field deserializes to. The default output name is the same as the input name.

Polymorphic Fields

Felds can refer to other models, such as an account with a user field:

```
class User(faust.Record):
    id: str
    first_name: str
    last_name: str

class Account(faust.Record):
    user: User
    balance: Decimal
```

This is a strict relationship: the value for Account.user can only be an instance of the `User` type.

Polymorphic fields are also supported, where the type of the field is decided at runtime.

Consider an Article models with a list of assets where the type of asset is decided at runtime:

```
class Asset(faust.Record):
    url: str
    type: str

class ImageAsset(Asset):
    type = 'image'

class VideoAsset(Asset):
    runtime_seconds: float
```

(continues on next page)

(continued from previous page)

```
    type = 'video'

class Article(faust.Record, polymorphic_fields=True):
    assets: List[Asset]
```

How does this work? Faust models add additional metadata when serialized, just look at the payload for one of our accounts:

```
>>> user = User(
...     id='07ecaebf-48c4-4c9e-92ad-d16d2f4a9a19',
...     first_name='Franz',
...     last_name='Kafka',
... )
>>> account = Account(
...     user=user,
...     balance='12.3',
)
>>> from pprint import pprint
>>> pprint(account.to_representation())
{
    '__faust': {'ns': 't.Account'},
    'balance': Decimal('12.3'),
    'user': {
        '__faust': {'ns': 't.User'},
        'first_name': 'Franz',
        'id': '07ecaebf-48c4-4c9e-92ad-d16d2f4a9a19',
        'last_name': 'Kafka',
    },
}
```

Here the metadata section is the `__faust` field, and it contains the name of the model that generated this payload.

By default we don't use this name for anything at all, but we do if polymorphic fields are enabled.

Why is it disabled by default? There is often a mismatch between the name of the class used to produce the event, and the class we want to reconstruct it as.

Imagine a producer is using an outdated version, or model cannot be shared between systems (this happens when using different programming languages, integrating with proprietary systems, and so on.)

The namespace `ns` contains the fully qualified name of the model class (in this example `t.User`).

Faust will keep an index of model names, and whenever you define a new model class we add it to this index.

Note: If you're trying to deserialize a model but it complains that it does not exist, you probably forgot to import this model before using it.

For the same reason you should not be renaming classes without having a strategy to do so in a forward compatible manner.

Validation

For models there is no validation of data by default: if you have a field described as an int, it will happily accept a string or any other object that you pass to it:

```
>>> class Person(faust.Record):
...     age: int
...

>>> p = Person(age="foo")
>>> p.age
"foo"
```

However, there is an option that will enable validation for all common JSON fields (`int`, `float`, `str`, etc.), and some commonly used Python ones (`datetime`, `Decimal`, etc.)

```
>>> class Person(faust.Record, validation=True):
...     age: int

>>> p = Person(age="foo")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValidationError: Invalid type for int field 'age': 'foo' (str)
```

For things like web forms raising an error automatically is not a good solution, as the client will usually want a list of all errors.

So in web views we suggest disabling automatic validation, and instead manually validating the model by calling `model.validate().` to get a list of `ValidationError` instances.

```
>>> class Person(faust.Record):
...     age: int
...     name: str

>>> p = Person(age="Gordon Gekko", name="32")
>>> p.validate()
[
  ('age': ValidationError(
       "Invalid type for int field 'age': 'Gordon Gekko' (str)"),
  ('name': ValidationError(
       "Invalid type for str field 'name': 32 (int)")),
]
```

Advanced Validation

If you have a field you want validation for, you may explicitly define the field descriptor for the field you want validation on (note: this will override the built-in validation for that field). This will also enable you to access more validation options, such as the maximum number of characters for a string, or a minmum value for an integer:

```
class Person(faust.Record, validation=True):
    age: int = IntegerField(min_value=18, max_value=99)
    name: str
```

Custom field types

You may define a custom `FieldDescriptor` subclass to perform your own validation:

```
from typing import Any, Iterable, List
from faust.exceptions import ValidationError
from faust.models import FieldDescriptor

class ChoiceField(FieldDescriptor[str]):

    def __init__(self, choices: List[str], **kwargs: Any) -> None:
        self.choices = choices
        # Must pass any custom args to init,
        # so we pass the choices keyword argument also here.
        super().__init__(choices=choices, **kwargs)

    def validate(self, value: str) -> Iterable[ValidationError]:
        if value not in self.choices:
            choices = ', '.join(self.choices)
            yield self.validation_error(
                f'{self.field} must be one of {choices}')
```

After defining the subclass you may use it to define model fields:

```
>>> class Order(faust.Record):
...     side: str = ChoiceField(['SELL', 'BUY'])

>>> Order(side='LEFT')
faust.exceptions.ValidationError: (
    'side must be one of SELL, BUY', <ChoiceField: Order.side: str>)
```

Excluding fields from representation

If you want your model to accept a certain field when deserializing, but exclude the same field from serialization, you can do so by marking that field as `exclude=True`:

```
import faust
from faust.models.fields import StringField


class Order(faust.Record):
    price: float
    quantity: float
    user_id: str = StringField(required=True, exclude=True)
```

This model will accept `user_id` as a keyword argument, and from any serialized structure:

```
>>> order = Order(price=30.0, quantity=2.0, user_id='foo')
>>> order.user_id
'foo'

>>> order2 = Order.loads(
...     '{"price": "30.0", quantity="2.0", "user_id": "foo"}',
...     serializer='json',
... )
```

(continues on next page)

(continued from previous page)

```
>>> order2.user_id
'foo'
```

But when serializing the order, the field will be excluded:

```
>>> order.asdict()
{'price': 30.0, 'quantity': 2.0}

>>> order.dumps(serializer='json')
'{"price": "30.0", "quantity": "2.0"}'
```

Reference

Serialization/Deserialization

class faust.**Record**

classmethod loads(*s: bytes*, ***, *default_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → faust.types.models.ModelT

Deserialize model object from bytes.

Keyword Arguments **serializer** (*CodecArg*) – Default serializer to use if no custom serializer was set for this model subclass.

Return type *ModelT*

dumps(***, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → bytes

Serialize object to the target serialization format.

Return type bytes

to_representation() → Mapping[str, Any]

Convert model to its Python generic counterpart.

Records will be converted to dictionary.

Return type Mapping[str, Any]

classmethod from_data(*data: Mapping*, ***, *preferred_type: Type[faust.types.models.ModelT] = None*) → faust.models.record.Record

Create model object from Python dictionary.

Return type *Record*

derive(**objects: faust.types.models.ModelT*, ***fields: Any*) → faust.types.models.ModelT

Derive new model with certain fields changed.

Return type *ModelT*

_options

Model metadata for introspection. An instance of *faust.types.models.ModelOptions*.

class faust.**ModelOptions**

fields = None

Flattened view of __annotations__ in MRO order.

Type Index

fieldset = None
Set of required field names, for fast argument checking.

Type Index

fieldpos = None
Positional argument index to field name. Used by Record.__init__ to map positional arguments to fields.

Type Index

optionalset = None
Set of optional field names, for fast argument checking.

Type Index

models = None
Mapping of fields that are ModelT

Type Index

coercions = None

defaults = None
Mapping of field names to default value.

Codecs

Supported codecs

- **raw** - no encoding/serialization (bytes only).
- **json** - `json` with UTF-8 encoding.
- **pickle** - `pickle` with Base64 encoding (not URL-safe).
- **binary** - Base64 encoding (not URL-safe).

Encodings are not URL-safe if the encoded payload cannot be embedded directly into a URL query parameter.

Serialization by name

The `dumps()` function takes a codec name and the object to encode as arguments, and returns bytes

```
>>> s = dumps('json', obj)
```

In reverse direction, the `loads()` function takes a codec name and an encoded payload to decode (in bytes), as arguments, and returns a reconstruction of the serialized object:

```
>>> obj = loads('json', s)
```

When passing in the codec type as a string (as in `loads('json', ...)` above), you can also combine multiple codecs to form a pipeline, for example `"json|gzip"` combines JSON serialization with gzip compression:

```
>>> obj = loads('json|gzip', s)
```

Codec registry

All codecs have a name and the *faust.serializers.codecs* attribute maintains a mapping from name to *Codec* instance.

You can add a new codec to this mapping by executing:

```
>>> from faust.serializers import codecs
>>> codecs.register(custom, custom_serializer())
```

To create a new codec, you need to define only two methods: first you need the `_loads()` method to deserialize bytes, then you need the `_dumps()` method to serialize an object:

```
import msgpack

from faust.serializers import codecs

class raw_msgpack(codecs.Codec):

    def _dumps(self, obj: Any) -> bytes:
        return msgpack.dumps(obj)

    def _loads(self, s: bytes) -> Any:
        return msgpack.loads(s)
```

We use `msgpack.dumps` to serialize, and our codec now encodes to raw msgpack format.

You may also combine the Base64 codec to support transports unable to handle binary data (such as HTTP or Redis):

Combining codecs is done using the `|` operator:

```
def msgpack() -> codecs.Codec:
    return raw_msgpack() | codecs.binary()

codecs.register('msgpack', msgpack())
```

```
>>> import my_msgpack_codec

>>> from faust import Record
>>> class Point(Record, serializer='msgpack'):
...     x: int
...     y: int
```

At this point we have to import the codec every time we want to use it, that is very cumbersome.

Faust also supports registering *codec extensions* using setuptools entry-points, so instead lets create an installable msgpack extension!

Define a package with the following directory layout:

```
faust-msgpack/
    setup.py
    faust_msgpack.py
```

The first file (`faust-msgpack/setup.py`) defines metadata about our package and should look like:

```
import setuptools

setuptools.setup(
```

(continues on next page)

(continued from previous page)

```
    name='faust-msgpack',
    version='1.0.0',
    description='Faust msgpack serialization support',
    author='Ola A. Normann',
    author_email='ola@normann.no',
    url='http://github.com/example/faust-msgpack',
    platforms=['any'],
    license='BSD',
    packages=find_packages(exclude=['ez_setup', 'tests', 'tests.*']),
    zip_safe=False,
    install_requires=['msgpack-python'],
    tests_require=[],
    entry_points={
        'faust.codecs': [
            'msgpack = faust_msgpack:msgpack',
        ],
    },
)
```

The most important part here is the `entry_points` section that tells setuptools how to load our plugin.

We have set the name of our codec to `msgpack` and the path to the codec class to be `faust_msgpack:msgpack`.

Faust imports this as it would do `from faust_msgpack import msgpack`, so we need to define hat part next in our `faust-msgpack/faust_msgpack.py` module:

```
from faust.serializers import codecs

class raw_msgpack(codecs.Codec):

    def _dumps(self, obj: Any) -> bytes:
        return msgpack.dumps(s)


def msgpack() -> codecs.Codec:
    return raw_msgpack() | codecs.binary()
```

That's it! To install and use our new extension do:

```
$ python setup.py install
```

At this point you can publish this to PyPI so it can be shared with other Faust users.

1.4.6 Tables and Windowing

> *"A man sees in the world what he carries in his heart."*
>
> – Goethe, *Faust: First part*

Tables

Basics

A table is a distributed in-memory dictionary, backed by a Kafka changelog topic used for persistence and fault-tolerance. We can replay the changelog upon network failure and node restarts, allowing us to rebuild the state of the table as it was before the fault.

To create a table use `app.Table`:

```
table = app.Table('totals', default=int)
```

You cannot modify a table outside of a stream operation; this means that you can only mutate the table from within an `async for event in stream:` block. We require this to align the table's partitions with the stream's, and to ensure the source topic partitions are correctly rebalanced to a different worker upon failure, along with any necessary table partitions.

Modifying a table outside of a stream will raise an error:

```
table = app.Table('totals', default=int)

# cannot modify table, as we are not iterating over stream
table['foo'] += 30
```

This source-topic-event to table-modification-event requirement also ensures that producing to the changelog and committing messages from the source happen simultaneously.

Warning: An abruptly terminated Faust worker can allow some changelog entries to go through, before having committed the source topic offsets.

Duplicate messages may result in double-counting and other data consistency issues, but since version 1.5 of Faust you can enable a setting for strict processing guarantees.

See the *`processing_guarantee`* setting for more information.

Co-partitioning Tables and Streams

When managing stream partitions and their corresponding changelog partitions, "co-partitioning" ensures the correct distribution of stateful processing among available clients, but one requirement is that tables and streams must share shards.

To shard the table differently, you must first repartition the stream using *group_by*.

Repartition a stream:

```
withdrawals_topic = app.topic('withdrawals', value_type=Withdrawal)

country_to_total = app.Table(
    'country_to_total', default=int).tumbling(10.0, expires=10.0)

withdrawals_stream = app.topic('withdrawals', value_type=Withdrawal).stream()
withdrawals_by_country = withdrawals_stream.group_by(Withdrawal.country)

@app.agent
async def process_withdrawal(withdrawals):
    async for withdrawal in withdrawals.group_by(Withdrawal.country):
        country_to_total[withdrawal.country] += withdrawal.amount
```

If the stream and table are not co-partitioned, we could end up with a table shard ending up on a different worker than the worker processing its corresponding stream partition.

Warning: For this reason, table changelog topics must have the same number of partitions as the source topic.

Table Sharding

Tables shards in Kafka must organize using a disjoint distribution of keys so that any computation for a subset of keys always happen together in the same worker process.

The following is an example of incorrect usage where subsets of keys are likely to be processed by different worker processes:

```
withdrawals_topic = app.topic('withdrawals', key_type=str,
                              value_type=Withdrawal)

user_to_total = app.Table('user_to_total', default=int)
country_to_total = app.Table(
    'country_to_total', default=int).tumbling(10.0, expires=10.0)


@app.agent(withdrawals_topic)
async def process_withdrawal(withdrawals):
    async for withdrawal in withdrawals:
        user_to_total[withdrawal.user] += withdrawal.amount
        country_to_total[withdrawal.country] += withdrawal.amount
```

Here the stream `withdrawals` is (implicitly) partitioned by the user ID used as message key. So the `country_to_total` table, instead of being partitioned by country name, is partitioned by the user ID. In practice, this means that data for a country may reside on multiple partitions, and worker instances end up with incomplete data.

To fix that rewrite your program like this, using two distinct agents and repartition the stream by country when populating the table:

```
withdrawals_topic = app.topic('withdrawals', value_type=Withdrawal)

user_to_total = app.Table('user_to_total', default=int)
country_to_total = app.Table(
    'country_to_total', default=int).tumbling(10.0, expires=10.0)


@app.agent(withdrawals_topic)
async def find_large_user_withdrawals(withdrawals):
    async for withdrawal in withdrawals:
        user_to_total[withdrawal.user] += withdrawal.amount


@app.agent(withdrawals_topic)
async def find_large_country_withdrawals(withdrawals):
    async for withdrawal in withdrawals.group_by(Withdrawal.country):
        country_to_total[withdrawal.country] += withdrawal.amount
```

The Changelog

Every modification to a table has a corresponding changelog update, the changelog is used to recover data after a failure.

We store the changelog in Kafka as a topic and use log compaction to only keep the *most recent value for a key in the log*. Kafka periodically compacts the table, to ensure the log does not grow beyond the number of keys in the table.

Note: In production the RocksDB store allows for almost instantaneous recovery of tables: a worker only needs to retrieve updates missed since last time the instance was up.

If you change the value for a key in the table, please make sure you update the table with the new value after:

In order to publish a changelog message into Kafka for fault-tolerance the table needs to be set explicitly. Hence, while changing values in Tables by reference, we still need to explicitly set the value to publish to the changelog, as shown below:

```
user_withdrawals = app.Table('user_withdrawals', default=list)
topic = app.topic('withdrawals', value_type=Withdrawal)

async for event in topic.stream():
    # get value for key in table
    withdrawals = user_withdrawals[event.account]
    # modify the value
    withdrawals.append(event.amount)
    # write it back to the table (also updating changelog)
    user_withdrawals[event.account] = withdrawals
```

If you forget to do so, like in the following example, the program will work but will have inconsistent data if a recovery is needed for any reason:

```
user_withdrawals = app.Table('user_withdrawals', default=list)
topic = app.topic('withdrawals', value_type=Withdrawal)

async for event in topic.stream():
    withdrawals = user_withdrawals[event.account]
    withdrawals.append(event.amount)
    # forgot to update the table with the new value
```

Due to this changelog, both table keys and values must be serializable.

See also:

- The *Models, Serialization, and Codecs* guide for more information about models and serialization.

Note: Faust creates an internal changelog topic for each table. The Faust application should be the only client producing to the changelog topics.

Windowing

Windowing allows us to process streams while preserving state over defined windows of time. A windowed table preserves key-value pairs according to the configured "Windowing Policy."

We support the following policies:

`class TumblingWindow`

This class creates fixed-sized, non-overlapping and contiguous time intervals to preserve key-value pairs, e.g. `Tumbling(10)` will create non-overlapping 10 seconds windows:

```
window 1: ----------
window 2:           ----------
window 3:                     ----------
window 4:                               ----------
window 5:                                         ----------
```

This class is exposed as a method from the output of `app.Table()`, it takes a mandatory parameter `size`, representing the window (time interval) duration and an optional parameter `expires`, representing the duration for which we want to store the data (key-value pairs) allocated to each window.

`class HoppingWindow`

This class creates fixed-sized, overlapping time intervals to preserve key-value pairs, e.g. `Hopping(10, 5)` will create overlapping 10 seconds windows. Each window will be created every 5 seconds.

```
window 1: ----------
window 2:      ----------
window 3:           ----------
window 4:                ----------
window 5:                     ----------
window 6:                          ----------
```

This class is exposed as a method from the output of `app.Table()`, it takes 2 mandatory parameters:

- `size`, representing the window (time interval) duration.
- `step`, representing the time interval used to create new windows.

It also takes an optional parameter `expires`, representing the duration for which we want to store the data (key-value pairs) allocated to each window.

How To

You can define a windowed table like this:

```
from datetime import timedelta
views = app.Table('views', default=int).tumbling(
    timedelta(minutes=1),
    expires=timedelta(hours=1),
)
```

Since a key can exist in multiple windows, the windowed table returns a special wrapper for `table[k]`, called a `WindowSet`.

Here's an example of a windowed table in use:

```
page_views_topic = app.topic('page_views', value_type=str)

@app.agent(events_topic)
async def aggregate_page_views(pages):
    # values in this stream are URLs as strings.
    async for page_url in pages:

        # increment one to all windows this page URL fall into.
        views[page_url] += 1

        if views[page_url].now() >= 10000:
            # Page is trending for current processing time window
            print('Trending now')

        if views[page_url].current() >= 10000:
            # Page would be trending in the current event's time window
            print('Trending when event happened')

        if views[page_url].value() >= 10000:
            # Page would be trending in the current event's time window
            # according to the relative time set when creating the
            # table.
            print('Trending when event happened')

        if views[page_url].delta(timedelta(minutes=30)) > views[page_url].now():
            print('Less popular compared to 30 minutes back')
```

In this table, `table[k].now()` returns the most recent value for the current processing window, overriding the _relative_to_ option used to create the window.

In this table, `table[k].current()` returns the most recent value relative to the time of the currently processing event, overriding the _relative_to_ option used to create the window.

In this table, `table[k].value()` returns the most recent value relative to the time of the currently processing event, and is the default behavior.

You can also make the current value relative to the current local time, relative to a different field in the event (if it has a custom timestamp field), or of another event.

The default behavior is "relative to current stream":

```
views = app.Table('views', default=int).tumbling(...).relative_to_stream()
```

Where `.relative_to_stream()` means values are selected based on the window of the current event in the currently processing stream.

You can also use `.relative_to_now()`: this means the window of the current local time is used instead:

```
views = app.Table('views', default=int).tumbling(...).relative_to_now()
```

If the current event has a custom timestamp field that you want to use, `relative_to_field(field_descriptor)` is suited for that task:

```
views = app.Table('views', default=int) \
    .tumbling(...) \
    .relative_to_field(Account.date_created)
```

You can override this default behavior when accessing data in the table:

```
@app.agent(topic)
async def process(stream):
    async for event in stream:
        # get latest value for 'key', based on the tables default
        # relative_to option.
        print(table[key].value())

        # You can bypass the default relative_to option, and
        # get the value closest to the event timestamp
        print(table[key].current())

        # You can bypass the default relative_to option, and
        # get the value closest to the current local time
        print(table[key].now())

        # Or get the value for a delta, e.g. 30 seconds ago, relative
        # to the event timestamp
        print(table[key].delta(30))
```

Note: We always retrieve window data based on timestamps. With tumbling windows there is just one window at a time, so for a given timestamp there is just one corresponding window. This is not the case for for hopping windows, in which a timestamp could be located in more than 1 window.

At this point, when accessing data from a hopping table, we always access the latest window for a given timestamp and we have no way of modifying this behavior.

Iterating over keys/values/items in a windowed table.

Note: Tables are distributed across workers, so when iterating over table contents you will only see the partitions assigned to the current worker.

Iterating over all the keys in a table will require you to visit all workers, which is highly impractical in a production system.

For this reason table iteration is mostly used in debugging and observing your system.

To iterate over the keys/items/values in windowed table you may add the `key_index` option to enable support for it:

```
windowed_table = app.Table(
    'name',
    default=int,
).hopping(10, 5, expires=timedelta(minutes=10), key_index=True)
```

Adding the key index means we keep a second table as an index of the keys present in the table. Whenever a new key is added we add the key to the key index, similarly whenever a key is deleted we also delete it from the index.

This enables fast iteration over the keys, items and values in the windowed table, with the caveat that those keys may not exist in all windows.

The table iterator views (`.keys()`/`.items()`/`.values()`) will be time-relative to the stream by default, unless you have changed the time-relativity using the `.relative_to_now` or `relative_to_timestamp` modifiers:

```
# Show keys present relative to time of current event in stream:
print(list(windowed_table.keys()))

# Show items present relative to time of current event in stream:
print(list(windowed_table.items()))

# Show values present relative to time of current event in stream:
print(list(windowed_table.values()))
```

You can also manually specify the time-relativity:

```
# Change time-relativity to current wall-clock time,
# and show a list of items present in that window.
print(list(windowed_table.relative_to_now().items()))

# Get items present 30 seconds ago:
print(list(windowed_table.relative_to_now().items().delta(30.0)))
```

"Out of Order" Events

Kafka maintains the order of messages published to it, but when using custom timestamp fields, relative ordering is not guaranteed.

For example, a producer can lose network connectivity while sending a batch of messages and be forced to retry sending them later, then messages in the topic won't be in timestamp order.

Windowed tables in Faust correctly handles such "out of order " events, at least until the message is as old as the table expiry configuration.

Note: We handle out of order events by storing separate aggregates for each window in the last `expires` seconds. The space complexity for this is `O(w * K)` where `w` is the number of windows in the last expires seconds and `K` is the number of keys in the table.

Table Serialization

A table is a mapping with keys and values, serialized using JSON by default.

If you want to use a different serialization mechanism you must configure that using the `key_serializer` and `value_serializer` arguments:

```
table = app.Table(
    'name',
    key_serializer='pickle',
    value_serializer='pickle',
)
```

1.4.7 Tasks, Timers, Cron Jobs, Web Views, and CLI Commands

Tasks

Your application will have agents that process events in streams, but can also start `asyncio.Task`-s that do other things, like periodic timers, views for the embedded web server, or additional command-line commands.

Decorating an async function with the *@app.task* decorator will tell the worker to start that function as soon as the worker is fully operational:

```
@app.task
async def on_started():
    print('APP STARTED')
```

If you add the above to the module that defines your app and start the worker, you should see the message printed in the output of the worker.

A task is a one-off task; if you want to do something at periodic intervals, you can use a timer.

Timers

A timer is a task that executes every `n` seconds:

```
@app.timer(interval=60.0)
async def every_minute():
    print('WAKE UP')
```

After starting the worker, and it's operational, the above timer will print something every minute.

Cron Jobs

A Cron job is a task that executes according to a Crontab format, usually at fixed times:

```
@app.crontab('0 20 * * *')
async def every_day_at_8_pm():
    print('WAKE UP ONCE A DAY')
```

After starting the worker, and it's operational, the above Cron job will print something every day at 8pm.

`crontab` takes 1 mandatory argument `cron_format` and 2 optional arguments:

- `tz`, represents the timezone. Defaults to None which gives behaves as UTC.
- `on_leader`, boolean defaults to False, only run on leader?

```
@app.crontab('0 20 * * *', tz=pytz.timezone('US/Pacific'), on_leader=True)
async def every_day_at_8_pm_pacific():
    print('WAKE UP AT 8:00pm PACIFIC TIME ONLY ON THE LEADER WORKER')
```

Web Views

The Faust worker will also expose a web server on every instance, that by default runs on port 6066. You can access this in your web browser after starting a worker instance on your local machine:

```
$ faust -A myapp worker -l info
```

Just point your browser to the local port to see statistics about your running instance:

```
http://localhost:6066
```

You can define additional views for the web server (called pages). The server will use the aiohttp HTTP server library, but you can also write custom web server drivers.

Add a simple page returning a JSON structure by adding this to your app module:

```
# this counter exists in-memory only,
# so will be wiped when the worker restarts.
count = [0]

@app.page('/count/')
async def get_count(self, request):
    # update the counter
    count[0] += 1
    # and return it.
    return self.json({
        'count': count[0],
    })
```

This example view is of limited usefulness. It only provides you with a count of how many times the page is requested, on that particular server, for as long as it's up, but you can also call actors or access table data in web views.

Restart your Faust worker, and you can visit your new page at:

```
http://localhost:6066/count/
```

Your workers may have an arbitrary number of views, and it's up to you what they provide. Just like other web applications they can communicate with Redis, SQL databases, and so on. Anything you want, really, and it's executing in an asynchronous event loop.

You can decide to develop your web app directly in the Faust workers, or you may choose to keep your regular web server separate from your Faust workers.

You can create complex systems quickly, just by putting everything in a single Faust app.

HTTP Verbs: GET/POST/PUT/DELETE

Specify a `faust.web.View` class when you need to handle HTTP verbs other than `GET`:

```
from faust.web import Request, Response, View

@app.page('/count/')
class counter(View):

    count: int = 0

    async def get(self, request: Request) -> Response
        return self.json({'count': self.count})

    async def post(self, request: Request) -> Response:
        n: int = request.query['n']
        self.count += 1
        return self.json({'count': self.count})

    async def delete(self, request: Request) -> Response:
        self.count = 0
```

Exposing Tables

A frequent requirement is the ability to expose table values in a web view, and while this is likely to be built-in to Faust in the future, you will have to implement this manually for now.

Tables are partitioned by key, and data for any specific key will exist on a particular worker instance. You can use the `@app.table_route` decorator to reroute the request to the worker holding that partition.

We define our table, and an agent reading from the stream to populate the table:

```
import faust

app = faust.App(
    'word-counts',
    broker='kafka://localhost:9092',
    store='rocksdb://',
    topic_partitions=8,
)

posts_topic = app.topic('posts', value_type=str)
word_counts = app.Table('word_counts', default=int,
                        help='Keep count of words (str to int).')


class Word(faust.Record):
    word: str

@app.agent(posts_topic)
async def shuffle_words(posts):
    async for post in posts:
        for word in post.split():
            await count_words.send(key=word, value=Word(word=word))

@app.agent()
async def count_words(words):
```

(continues on next page)

(continued from previous page)

```
    """Count words from blog post article body."""
    async for word in words:
        word_counts[word.word] += 1
```

After that we define the view, using the `@app.table_route` decorator to reroute the request to the correct worker instance:

```
@app.page('/count/{word}/')
@app.table_route(table=word_counts, match_info='word')
async def get_count(web, request, word):
    return web.json({
        word: word_counts[word],
    })
```

In the above example we used part of the URL to find the given word, but you may also want to get this from query parameters.

Table route based on key in query parameter:

```
@app.page('/count/')
@app.table_route(table=word_counts, query_param='word')
async def get_count(web, request):
    word = request.query['word']
    return web.json({
        word: word_counts[word],
    })
```

CLI Commands

As you may already know, you can make your project into an executable, that can start Faust workers, list agents, models and more, just by calling `app.main()`.

Even if you don't do that, the **`faust`** program is always available and you can point it to any app:

```
$ faust -A myapp worker -l info
```

The `myapp` argument should point to a Python module/package having an `app` attribute. If the attribute has a different name, please specify a fully qualified path:

```
$ faust -A myproj.apps:faust_app worker -l info
```

Do `--help` to get a list of subcommands supported by the app:

```
$ faust -A myapp --help
```

To turn your script into the **`faust`** command, with the *`-A`* option already set, add this to the end of the module:

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

If saved as `simple.py` you can now execute it as if it was the **`faust`** program:

```
$ python simple.py worker -l info
```

Custom CLI Commands

To add a custom command to your app, see the `examples/simple.py` example in the Faust distribution, where we added a `produce` command used to send example data into the stream processors:

```
from faust.cli import option

# the full example is in examples/simple.py in the Faust distribution.
# this only shows the command part of this code.

@app.command(
    option('--max-latency',
           type=float, default=PRODUCE_LATENCY,
           help='Add delay of (at most) n seconds between publishing.'),
    option('--max-messages',
           type=int, default=None,
           help='Send at most N messages or 0 for infinity.'),
)
async def produce(self, max_latency: float, max_messages: int):
    """Produce example Withdrawal events."""
    num_countries = 5
    countries = [f'country_{i}' for i in range(num_countries)]
    country_dist = [0.9] + ([0.10 / num_countries] * (num_countries - 1))
    num_users = 500
    users = [f'user_{i}' for i in range(num_users)]
    self.say('Done setting up. SENDING!')
    for i in range(max_messages) if max_messages is not None else count():
        withdrawal = Withdrawal(
            user=random.choice(users),
            amount=random.uniform(0, 25_000),
            country=random.choices(countries, country_dist)[0],
            date=datetime.utcnow().replace(tzinfo=timezone.utc),
        )
        await withdrawals_topic.send(key=withdrawal.user, value=withdrawal)
        if not i % 10000:
            self.say(f'+SEND {i}')
        if max_latency:
            await asyncio.sleep(random.uniform(0, max_latency))
```

The `@app.command` decorator accepts both `click.option` and `click.argument`, so you can specify command-line options, as well as command-line positional arguments.

Daemon Commands

The `daemon` flag can be set to mark the command as a background service that won't exit until the user hits `Control-c`, or the process is terminated by another signal:

```
@app.command(
    option('--foo', type=float, default=1.33),
    daemon=True,
)
async def my_daemon(self, foo: float):
    print('STARTING DAEMON')
    ...
    # set up some stuff
    # we can return here but the program will not shut down
```

(continues on next page)

(continued from previous page)

```
    # until the user hits ^Cbdb Control-C , or the process is terminated
    # by signal
    return
```

1.4.8 Command-line Interface

- *Program:* `faust`
  - `faust --version` - *Show version information and exit.*
  - `faust --help` - *Show help and exit.*
  - `faust agents` - *List agents defined in this application.*
  - `faust models` - *List defined serialization models.*
  - `faust model <name>` - *List model fields by model name.*
  - `faust reset` - *Delete local table state.*
  - `faust send <topic/agent> <message_value>` - *Send message.*
  - `faust tables` - *List Tables (distributed K/V stores).*
  - `faust worker` - *Start Faust worker instance.*

Program: `faust`

The **`faust`** umbrella command hosts all command-line functionality for Faust. Projects may add custom commands using the `@app.command` decorator (see *CLI Commands*).

Options:

`-A, --app`
Path of Faust application to use, or the name of a module.

`--quiet, --no-quiet, -q`
Silence output to <stdout>/<stderr>.

`--debug, --no-debug`
Enable debugging output, and the blocking detector.

`--workdir, -W`
Working directory to change to after start.

`--datadir`
Directory to keep application state.

`--json`
Return output in machine-readable JSON format.

`--loop, -L`
Event loop implementation to use: `aio` (default), `eventlet`, `uvloop`.

Why is `examples/word_count.py` used as the program?

The convention for Faust projects is to define an entry point for the Faust command using `app.main()` - see *app.main() – Start the faust command-line program.* to see how to do so.

For a standalone program such as `examples/word_count.py` this is accomplished by adding the following at the end of the file:

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

For a project organized in modules (a package) you can add a `package/__main__.py` module:

```
# package/__main__.py
from package.app import app
app.main()
```

Or use setuptools entry points so that `pip install myproj` installs a command-line program.

Even if you don't add an entry point you can always use the **`faust`** program by specifying the path to an app.

Either the name of a module having an `app` attribute:

```
$ faust -A examples.word_count
```

or specifying the attribute directly:

```
$ faust -A examples.word_count:app
```

`faust --version` - Show version information and exit.

Example:

```
$ python examples/word_count.py --version
word_count.py, version Faust 0.9.39
```

`faust --help` - Show help and exit.

Example:

```
$ python examples/word_count.py --help
Usage: word_count.py [OPTIONS] COMMAND [ARGS]...

Faust command-line interface.

Options:
-L, --loop [aio|eventlet|uvloop]
                                Event loop implementation to use.
--json / --no-json              Prefer data to be emitted in json format.
-D, --datadir DIRECTORY         Directory to keep application state.
-W, --workdir DIRECTORY         Working directory to change to after start.
--no-color / --color            Enable colors in output.
--debug / --no-debug            Enable debugging output, and the blocking
                                detector.
-q, --quiet / --no-quiet        Silence output to <stdout>/<stderr>.
-A, --app TEXT                  Path of Faust application to use, or the
```

(continues on next page)

(continued from previous page)

```
                                name of a module.
--version                       Show the version and exit.
--help                          Show this message and exit.

Commands:
agents  List agents.
model   Show model detail.
models  List all available models as tabulated list.
reset   Delete local table state.
send    Send message to agent/topic.
tables  List available tables.
worker  Start faust worker instance.
```

`faust agents` - List agents defined in this application.

Example:

```
$ python examples/word_count.py agents
┌Agents──────────┬──────────────────────────────────────────────────┬──────────────────
│ name           │ topic                                            │ help             ␣
↪                 │
├────────────────┼──────────────────────────────────────────────────┼──────────────────
│ @count_words   │ word-counts-examples.word_count.count_words      │ Count words from␣
↪blog post article body. │
│ @shuffle_words │ posts                                            │ <N/A>            ␣
↪                 │
└────────────────┴──────────────────────────────────────────────────┴──────────────────
```

JSON Output using `--json`:

```
$ python examples/word_count.py --json agents
[{"name": "@count_words",
  "topic": "word-counts-examples.word_count.count_words",
  "help": "Count words from blog post article body."},
 {"name": "@shuffle_words",
  "topic": "posts",
  "help": "<N/A>"}]
```

`faust models` - List defined serialization models.

Example:

```
$ python examples/word_count.py models
┌Models┬────────┐
│ name │ help   │
├──────┼────────┤
│ Word │ <N/A>  │
└──────┴────────┘
```

JSON Output using `--json`:

```
python examples/word_count.py --json models
[{"name": "Word", "help": "<N/A>"}]
```

`faust model <name>` - List model fields by model name.

Example:

```
$ python examples/word_count.py model Word
┌Word─────┬─────────┬──────────┐
│ field   │ type    │ default* │
├─────────┼─────────┼──────────┤
│ word    │ str     │ *        │
└─────────┴─────────┴──────────┘
```

JSON Output using `--json`:

```
$ python examples/word_count.py --json model Word
[{"field": "word", "type": "str", "default*": "*"}]
```

`faust reset` - Delete local table state.

Warning: This command will result in the destruction of the following files:

1) **The local database directories/files backing tables** (does not apply if an in-memory store like memory:// is used).

Note: This data is technically recoverable from the Kafka cluster (if intact), but it'll take a long time to get the data back as you need to consume each changelog topic in total.

It'd be faster to copy the data from any standbys that happen to have the topic partitions you require.

Example:

```
$ python examples/word_count.py reset
```

`faust send <topic/agent> <message_value>` - Send message.

Options:

`--key-type, -K`
Name of model to serialize key into.

`--key-serializer`
Override default serializer for key.

`--value-type, -V`
Name of model to serialize value into.

`--value-serializer`
Override default serializer for value.

`--key, -k`
String value for key (use json if model).

`--partition`
Specific partition to send to.

--repeat, -r
Send message n times.

--min-latency
Minimum delay between sending.

--max-latency
Maximum delay between sending.

Examples:

Send to agent by name using `@` prefix:

```
$ python examples/word_count.py send @word_count "foo"
```

Send to topic by name (no prefix):

```
$ python examples/word_count.py send mytopic "foo"
{"topic": "mytopic",
 "partition": 2,
 "topic_partition": ["mytopic", 2],
 "offset": 0,
 "timestamp": 1520974493620,
 "timestamp_type": 0}
```

To get help:

```
$ python examples/word_count.py send --help
Usage: word_count.py send [OPTIONS] ENTITY [VALUE]

Send message to agent/topic.

Options:
-K, --key-type TEXT      Name of model to serialize key into.
--key-serializer TEXT    Override default serializer for key.
-V, --value-type TEXT    Name of model to serialize value into.
--value-serializer TEXT  Override default serializer for value.
-k, --key TEXT           String value for key (use json if model).
--partition INTEGER      Specific partition to send to.
-r, --repeat INTEGER     Send message n times.
--min-latency FLOAT      Minimum delay between sending.
--max-latency FLOAT      Maximum delay between sending.
--help                   Show this message and exit.
```

`faust tables` - List Tables (distributed K/V stores).

Example:

```
$ python examples/word_count.py tables
┌Tables──────┬──────────────────────────────────────┐
│ name       │ help                                 │
├────────────┼──────────────────────────────────────┤
│ word_counts │ Keep count of words (str to int). │
└────────────┴──────────────────────────────────────┘
```

JSON Output using `--json`:

```
$ python examples/word_count.py --json tables
[{"name": "word_counts", "help": "Keep count of words (str to int)."}]
```

`faust worker` - Start Faust worker instance.

A "worker" starts a single instance of a Faust application.

Options:

`--logfile, -f`
Path to logfile (default is <stderr>).

`--loglevel, -l`
Logging level to use: `CRIT|ERROR|WARN|INFO|DEBUG`.

`--blocking-timeout`
Blocking detector timeout (requires –debug).

`--without-web`
Do not start embedded web server.

`--web-host, -h`
Canonical host name for the web server.

`--web-port, -p`
Port to run web server on (default is 6066).

`--web-bind, -b`
Network mask to bind web server to (default is "0.0.0.0" - all interfaces).

`--console-port`
When *`faust --debug`* is enabled this specifies the port to run the aiomonitor console on (default is 50101).

Examples:

```
$ python examples/word_count.py worker
┌ƒaµS† v1.0.0─────────────────────────────────────┐
| id          | word-counts                       |
| transport   | kafka://localhost:9092            |
| store       | rocksdb:                          |
| web         | http://localhost:6066/            |
| log         | -stderr- (warn)                   |
| pid         | 46052                             |
| hostname    | grainstate.local                  |
| platform    | CPython 3.6.4 (Darwin x86_64)     |
| drivers     | aiokafka=0.4.0 aiohttp=3.0.8      |
| datadir     | /opt/devel/faust/word-counts-data |
| appdir      | /opt/devel/faust/word-counts-data/v1 |
└─────────────┴───────────────────────────────────┘
starting
```

To get more logging use *-l info* (or further *-l debug*):

```
$ python examples/word_count.py worker -l info
┌ƒaµS† v1.0.0─────────────────────────────────────┐
| id          | word-counts                       |
| transport   | kafka://localhost:9092            |
| store       | rocksdb:                          |
```

(continues on next page)

(continued from previous page)

```
| web        | http://localhost:6066/                    |
| log        | -stderr- (info)                           |
| pid        | 46034                                     |
| hostname   | grainstate.local                          |
| platform   | CPython 3.6.4 (Darwin x86_64)             |
| drivers    | aiokafka=0.4.0 aiohttp=3.0.8              |
| datadir    | /opt/devel/faust/word-counts-data         |
| appdir     | /opt/devel/faust/word-counts-data/v1      |
└------------┴-------------------------------------------┘
starting^[2018-03-13 13:41:39,269: INFO]: [^Worker]: Starting...
[2018-03-13 13:41:39,275: INFO]: [^-App]: Starting...
[2018-03-13 13:41:39,271: INFO]: [^--Web]: Starting...
[2018-03-13 13:41:39,272: INFO]: [^---ServerThread]: Starting...
[2018-03-13 13:41:39,273: INFO]: [^--Web]: Serving on http://localhost:6066/
[2018-03-13 13:41:39,275: INFO]: [^--Monitor]: Starting...
[2018-03-13 13:41:39,275: INFO]: [^--Producer]: Starting...
[2018-03-13 13:41:39,317: INFO]: [^--Consumer]: Starting...
[2018-03-13 13:41:39,325: INFO]: [^--LeaderAssignor]: Starting...
[2018-03-13 13:41:39,325: INFO]: [^--Producer]: Creating topic word-counts-__assignor-
↪__leader
[2018-03-13 13:41:39,325: INFO]: [^--Producer]: Nodes: [0]
[2018-03-13 13:41:39,668: INFO]: [^--Producer]: Topic word-counts-__assignor-__leader␣
↪created.
[2018-03-13 13:41:39,669: INFO]: [^--ReplyConsumer]: Starting...
[2018-03-13 13:41:39,669: INFO]: [^--Agent]: Starting...
[2018-03-13 13:41:39,673: INFO]: [^---OneForOneSupervisor]: Starting...
[2018-03-13 13:41:39,673: INFO]: [^---Agent*: examples.word_co[.]shuffle_words]:␣
↪Starting...
[2018-03-13 13:41:39,673: INFO]: [^--Agent]: Starting...
[2018-03-13 13:41:39,674: INFO]: [^---OneForOneSupervisor]: Starting...
[2018-03-13 13:41:39,674: INFO]: [^---Agent*: examples.word_count.count_words]:␣
↪Starting...
[2018-03-13 13:41:39,674: INFO]: [^--Conductor]: Starting...
[2018-03-13 13:41:39,674: INFO]: [^--TableManager]: Starting...
[2018-03-13 13:41:39,675: INFO]: [^--Stream: <(*)Topic: posts@0x10497e5f8>]: Starting.
↪..
[2018-03-13 13:41:39,675: INFO]: [^--Stream: <(*)Topic: wo...s@0x105f73b38>]:␣
↪Starting...
[...]
```

To get help use `faust worker --help`:

```
$ python examples/word_count.py worker --help
Usage: word_count.py worker [OPTIONS]

Start faust worker instance.

Options:
-f, --logfile PATH              Path to logfile (default is <stderr>).
-l, --loglevel [crit|error|warn|info|debug]
                                Logging level to use.
--blocking-timeout FLOAT        Blocking detector timeout (requires
                                --debug).
-p, --web-port RANGE[1-65535]   Port to run web server on.
-b, --web-bind TEXT
-h, --web-host TEXT             Canonical host name for the web server.
--console-port RANGE[1-65535]   (when --debug:) Port to run debugger console
```

(continues on next page)

(continued from previous page)

```
                                on.
--help                          Show this message and exit.
```

1.4.9 Sensors - Monitors and Statistics

Basics

Sensors record information about events occurring in a Faust application as they happen.

There's a default sensor called "the monitor" that record the runtime of messages and events as they go through the worker, the latency of publishing messages, the latency of committing Kafka offsets, and so on.

The web server uses this monitor to present graphs and statistics about your instance, and there's also a versions of the monitor available that forwards statistics to StatsD, and Datadog.

You can define custom sensors to record the information that you care about, and enable them in the worker.

Monitor

The *faust.Monitor* is a built-in sensor that captures information like:

- Average message processing time (when all agents have processed a message).
- Average event processing time (from an event received by an agent to the event is *acked*.)
- The total number of events processed every second.
- The total number of events processed every second listed by topic.
- The total number of events processed every second listed by agent.
- The total number of records written to tables.
- Duration of Kafka topic commit operations (latency).
- Duration of producing messages (latency).

You can access the state of the monitor, while the worker is running, in `app.monitor`:

```
@app.agent(app.topic('topic'))
def mytask(events):
    async for event in events:
        # print how many events are being processed every second.
        print(app.monitor.events_s)
```

Monitor API Reference

Class: `Monitor`

Monitor Attributes

class faust.**Monitor**

messages_active
Number of messages currently being processed.

messages_received_total
Number of messages processed in total.

messages_received_by_topic
Count of messages received by topic

messages_s
Number of messages being processed this second.

events_active
Number of events currently being processed.

events_total
Number of events processed in total.

events_s
Number of events being processed this second.

events_by_stream
Count of events processed by stream

events_by_task
Count of events processed by task

events_runtime
Deque of run times used for averages

events_runtime_avg
Average event runtime over the last second.

tables
Mapping of tables

commit_latency
Deque of commit latency values

send_latency
Deque of send latency values

messages_sent
Number of messages sent in total.

send_errors
Number of produce operations that ended in error.

messages_sent_by_topic
Number of messages sent by topic.

topic_buffer_full
Counter of times a topics buffer was full

metric_counts
Arbitrary counts added by apps

tp_committed_offsets
Last committed offsets by TopicPartition

tp_read_offsets
Last read offsets by TopicPartition

tp_end_offsets
Log end offsets by TopicPartition

assignment_latency
Deque of assignment latency values.

assignments_completed
Number of partition assignments completed.

assignments_failed
Number of partitions assignments that failed.

rebalances
Number of rebalances seen by this worker.

rebalance_return_latency
Deque of previous n rebalance return latencies.

rebalance_end_latency
Deque of previous n rebalance end latencies.

rebalance_return_avg
Average rebalance return latency.

rebalance_end_avg
Average rebalance end latency.

http_response_codes
Counter of returned HTTP status codes.

http_response_latency
Deque of previous n HTTP request->response latencies.

http_response_latency_avg
Average request->response latency.

Configuration Attributes

class faust.**Monitor**

max_avg_history = 100
Max number of total run time values to keep to build average.

max_commit_latency_history = 30
Max number of commit latency numbers to keep.

max_send_latency_history = 30
Max number of send latency numbers to keep.

Class: `TableState`

class faust.sensors.**TableState**

TableState.**table = None**

TableState.**keys_retrieved = 0**
Number of times a key has been retrieved from this table.

TableState.**keys_updated = 0**
Number of times a key has been created/changed in this table.

TableState.**keys_deleted = 0**
Number of times a key has been deleted from this table.

Sensor API Reference

This reference describes the sensor interface and is useful when you want to build custom sensors.

Methods

Message Callbacks

class faust.**Sensor**

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Message received by a consumer.
Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
All streams finished processing message.
Return type None

Event Callbacks

class faust.**Sensor**

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Message sent to a stream as an event.
Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Event was acknowledged by stream.

Notes

Acknowledged means a stream finished processing the event, but given that multiple streams may be handling the same event, the message cannot be committed before all streams have processed it. When all streams have acknowledged the event, it will go through `on_message_out()` just before offsets are committed.

Return type None

Table Callbacks

class faust.**Sensor**

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Key retrieved from table.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Value set for key in table.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Key deleted from table.

Return type None

Consumer Callbacks

class faust.**Sensor**

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Consumer is about to commit topic offset.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Consumer finished committing topic offset.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None
Topic buffer full so conductor had to wait.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

Producer Callbacks

class faust.**Sensor**

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
About to send a message.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Message successfully sent.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Error while sending message.

Return type None

Web Callbacks

class faust.**Sensor**

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, ***, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT, request: faust.web.base.Request, response: Optional[faust.web.base.Response], state: Dict, *, view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

1.4.10 Testing

Basics

To test an agent when unit testing or functional testing, use the special `Agent.test()` mode to send items to the stream while processing it locally:

```
app = faust.App('test-example')

class Order(faust.Record, serializer='json'):
    account_id: str
    product_id: str
    amount: int
    price: float

orders_topic = app.topic('orders', value_type=Order)
orders_for_account = app.Table('order-count-by-account', default=int)

@app.agent(orders_topic)
async def order(orders):
    async for order in orders.group_by(Order.account_id):
        orders_for_account[order.account_id] += 1
        yield order
```

Our agent reads a stream of orders and keeps a count of them by account id in a distributed table also partitioned by the account id.

To test this agent we use `order.test_context()`:

```
async def test_order():
    # start and setup the agent in this block
    async with order.test_context() as agent:
        order = Order(account_id='1', product_id='2', amount=1, price=300)
        # send order to the test agents local channel, and wait
        # for the agent to process it.
        await agent.put(order)
        # at this point the agent already updated the table
        assert orders_for_account[order.account_id] == 1
        await agent.put(order)
        assert orders_for_account[order.account_id] == 2

async def run_tests():
    app.conf.store = 'memory://'  # tables must be in-memory
    await test_order()

if __name__ == '__main__':
    import asyncio
```

(continues on next page)

(continued from previous page)

```
    loop = asyncio.get_event_loop()
    loop.run_until_complete(run_tests())
```

For the rest of this guide we'll be using pytest and pytest-asyncio for our examples. If you're using a different testing framework you may have to adapt them a bit to work.

Testing with pytest

Testing that an agent sends to topic/calls another agent.

When unit testing you should mock any dependencies of the agent being tested,

- If your agent calls another function: mock that function to verify it was called.
- If your agent sends a message to a topic: mock that topic to verify a message was sent.
- If your agent calls another agent: mock the other agent to verify it was called.

Here's an example agent that calls another agent:

```
import faust

app = faust.App('example-test-agent-call')

@app.agent()
async def foo(stream):
    async for value in stream:
        await bar.send(value)
        yield value

@app.agent()
async def bar(stream):
    async for value in stream:
        yield value + 'YOLO'
```

To test these two agents you have to test them in isolation of each other: first test `foo` with `bar` mocked, then in a different test do `bar`:

```
import pytest
from unittest.mock import Mock, patch

from example import app, foo, bar

@pytest.fixture()
def test_app(event_loop):
    """passing in event_loop helps avoid 'attached to a different loop' error"""
    app.finalize()
    app.conf.store = 'memory://'
    app.flow_control.resume()
    return app

@pytest.mark.asyncio()
async def test_foo(test_app):
    with patch(__name__ + '.bar') as mocked_bar:
        mocked_bar.send = mock_coro()
        async with foo.test_context() as agent:
```

(continues on next page)

(continued from previous page)

```
            await agent.put('hey')
            mocked_bar.send.assert_called_with('hey')

def mock_coro(return_value=None, **kwargs):
    """Create mock coroutine function."""
    async def wrapped(*args, **kwargs):
        return return_value
    return Mock(wraps=wrapped, **kwargs)

@pytest.mark.asyncio()
async def test_bar(test_app):
    async with bar.test_context() as agent:
        event = await agent.put('hey')
        assert agent.results[event.message.offset] == 'heyYOLO'
```

Note: The pytest-asyncio extension must be installed to run these tests. If you don't have it use **`pip`** to install it:

```
$ pip install -U pytest-asyncio
```

Testing and windowed tables

If your table is windowed and you want to verify that the value for a key is correctly set, use `table[k].current(event)` to get the value placed within the window of the current event:

```
import faust
import pytest

@pytest.mark.asyncio()
async def test_process_order():
    app.conf.store = 'memory://'
    async with process_order.test_context() as agent:
        order = Order(account_id='1', product_id='2', amount=1, price=300)
        event = await agent.put(order)

        # windowed table: we select window relative to the current event
        assert orders_for_account['1'].current(event) == 1

        # in the window 3 hours ago there were no orders:
        assert orders_for_account['1'].delta(3600 * 3, event)


class Order(faust.Record, serializer='json'):
    account_id: str
    product_id: str
    amount: int
    price: float

app = faust.App('test-example')
orders_topic = app.topic('orders', value_type=Order)

# order count within the last hour (window is a 1-hour TumblingWindow)
orders_for_account = app.Table(
    'order-count-by-account', default=int,
```

(continues on next page)

(continued from previous page)

```
).tumbling(3600).relative_to_stream()

@app.agent(orders_topic)
async def process_order(orders):
    async for order in orders.group_by(Order.account_id):
        orders_for_account[order.account_id] += 1
        yield order
```

1.4.11 LiveCheck: End-to-end test for production/staging.

What is the problem with unit tests? What is difficult about maintaining integration tests? Why testing in production makes sense. Bucket testing, slow deploy of new tests will give confidence of a release. A staging environment is still desirable.

Enables you to:

- track requests as they travel through your micro service architecture.
- define contracts that should be met at every step

all by writing a class that looks like a regular unit test.

This is a passive observer, so will be able to detect and complain when subsystems are down. Tests are executed based on probability, so you can run tests for every requests, or for just 30%, 50%, or even 0.1% of your requests.

This is not just for micro service architectures, it's for any asynchronous system. A monolith sending celery tasks is a good example, you could track and babysit at every step of a work flow to make sure things progress as they should.

Every stage of your production pipeline could be tested for things such as "did the account debt exceed a threshold after this change", or "did the account earn a lot of credits after this change".

This means LiveCheck can be used to monitor and alert on anomalies happening in your product, as well as for testing general reliability and the consistency of your distributed system.

Every LiveCheck test case is a stream processor, and so can use tables to store data.

Tutorial

LiveCheck Example.

1) First start an instance of the stock ordering system in a new terminal:

```
$ python examples/livecheck.py worker -l info
```

2) Then in a new terminal, start a LiveCheck instance for this app

```
$ python examples/livecheck.py livecheck -l info
```

3) Then visit `http://localhost:6066/order/init/sell/` in your browser.

   Alternatively you can use the `post_order` command:

```
$ python examples/livecheck.py post_order --side=sell
```

The probability of a test execution happening is 50% so you have to do this at least twice to see activity happening in the LiveCheck instance terminal.

1.4.12 Configuration Reference

Required Settings

id

type `str`

A string uniquely identifying the app, shared across all instances such that two app instances with the same *id* are considered to be in the same "group".

This parameter is required.

The id and Kafka

When using Kafka, the id is used to generate app-local topics, and names for consumer groups.

Commonly Used Settings

debug

type `bool`

default `False`

Use in development to expose sensor information endpoint.

Tip: If you want to enable the sensor statistics endpoint in production, without enabling the *debug* setting, you can do so by adding the following code:

```
app.web.blueprints.add('/stats/', 'faust.web.apps.stats:blueprint')
```

---

broker

type `Union[str, URL, List[URL]]`

default `[URL("kafka://localhost:9092")]`

Faust needs the URL of a "transport" to send and receive messages.

Currently, the only supported production transport is `kafka://`. This uses the aiokafka client under the hood, for consuming and producing messages.

You can specify multiple hosts at the same time by separating them using the semi-comma:

```
kafka://kafka1.example.com:9092;kafka2.example.com:9092
```

Which in actual code looks like this:

```
app = faust.App(
    'id',
    broker='kafka://kafka1.example.com:9092;kafka2.example.com:9092',
)
```

You can also pass a list of URLs:

```
app = faust.App(
    'id',
    broker=['kafka://kafka1.example.com:9092',
            'kafka://kafka2.example.com:9092'],
)
```

See also:

You can configure the transport used for consuming and producing separately, by setting the *broker_consumer* and *broker_producer* settings.

This setting is used as the default.

Available Transports

- `kafka://`

  Alias to `aiokafka://`

- `aiokafka://`

  The recommended transport using the aiokafka client.

  Limitations: None

- `confluent://`

  Experimental transport using the confluent-kafka client.

  Limitations: Does not do sticky partition assignment (not suitable for tables), and do not create any necessary internal topics (you have to create them manually).

broker_credentials

New in version 1.5.

type *CredentialsT*

default `None`

Specify the authentication mechanism to use when connecting to the broker.

The default is to not use any authentication.

SASL Authentication

You can enable SASL authentication via plain text:

```
app = faust.App(
    broker_credentials=faust.SASLCredentials(
        username='x',
        password='y',
    ))
```

Warning: Do not use literal strings when specifying passwords in production, as they can remain visible in stack traces.

Instead the best practice is to get the password from a configuration file, or from the environment:

```
BROKER_USERNAME = os.environ.get('BROKER_USERNAME')
BROKER_PASSWORD = os.environ.get('BROKER_PASSWORD')

app = faust.App(
    broker_credentials=faust.SASLCredentials(
        username=BROKER_USERNAME,
        password=BROKER_PASSWORD,
    ))
```

GSSAPI Authentication

GSSAPI authentication over plain text:

```
app = faust.App(
    broker_credentials=faust.GSSAPICredentials(
        kerberos_service_name='faust',
        kerberos_domain_name='example.com',
    ),
)
```

GSSAPI authentication over SSL:

```
import ssl
ssl_context = ssl.create_default_context(
    purpose=ssl.Purpose.SERVER_AUTH, cafile='ca.pem')
ssl_context.load_cert_chain('client.cert', keyfile='client.key')

app = faust.App(
    broker_credentials=faust.GSSAPICredentials(
        kerberos_service_name='faust',
        kerberos_domain_name='example.com',
        ssl_context=ssl_context,
    ),
)
```

SSL Authentication

Provide an SSL context for the Kafka broker connections.

This allows Faust to use a secure SSL/TLS connection for the Kafka connections and enabling certificate-based authentication.

```
import ssl

ssl_context = ssl.create_default_context(
    purpose=ssl.Purpose.SERVER_AUTH, cafile='ca.pem')
ssl_context.load_cert_chain('client.cert', keyfile='client.key')
app = faust.App(..., broker_credentials=ssl_context)
```

store

type `str`

default `URL("memory://")`

The backend used for table storage.

Tables are stored in-memory by default, but you should not use the `memory://` store in production.

In production, a persistent table store, such as `rocksdb://` is preferred.

cache

New in version 1.2.

type `str`

default `URL("memory://")`

Optional backend used for Memcached-style caching. URL can be: `redis://host`, `rediscluster://host`, or `memory://`.

processing_guarantee

New in version 1.5.

type `str`

default `"at_least_once"`

The processing guarantee that should be used.

Possible values are "at_least_once" (default) and "exactly_once". Note that if exactly-once processing is enabled consumers are configured with `isolation.level="read_committed"` and producers are configured with `retries=Integer.MAX_VALUE` and `enable.idempotence=true` per default. Note that by default exactly-once processing requires a cluster of at least three brokers what is the recommended setting for production. For development you can change this, by adjusting broker setting `transaction.state.log.replication.factor` to the number of brokers you want to use.

autodiscover

type `Union[bool, Iterable[str], Callable[[], Iterable[str]]]`

Enable autodiscovery of agent, task, timer, page and command decorators.

Faust has an API to add different `asyncio` services and other user extensions, such as "Agents", HTTP web views, command-line commands, and timers to your Faust workers. These can be defined in any module, so to discover them at startup, the worker needs to traverse packages looking for them.

Warning: The autodiscovery functionality uses the Venusian library to scan wanted packages for `@app.agent`, `@app.page`, `@app.command`, `@app.task` and `@app.timer` decorators, but to do so, it's required to traverse the package path and import every module in it.

Importing random modules like this can be dangerous so make sure you follow Python programming best practices. Do not start threads; perform network I/O; do test monkey-patching for mocks or similar, as a side effect of importing a module. If you encounter a case such as this then please find a way to perform your action in a lazy manner.

Warning: If the above warning is something you cannot fix, or if it's out of your control, then please set `autodiscover=False` and make sure the worker imports all modules where your decorators are defined.

The value for this argument can be:

`bool` If `App(autodiscover=True)` is set, the autodiscovery will scan the package name described in the `origin` attribute.

The `origin` attribute is automatically set when you start a worker using the **`faust`** command line program, for example:

```
faust -A example.simple worker
```

The `-A`, option specifies the app, but you can also create a shortcut entry point by calling `app.main()`:

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

Then you can start the **`faust`** program by executing for example `python myscript.py worker --loglevel=INFO`, and it will use the correct application.

`Sequence[str]` The argument can also be a list of packages to scan:

```
app = App(..., autodiscover=['proj_orders', 'proj_accounts'])
```

`Callable[[], Sequence[str]]` The argument can also be a function returning a list of packages to scan:

```
def get_all_packages_to_scan():
    return ['proj_orders', 'proj_accounts']

app = App(..., autodiscover=get_all_packages_to_scan)
```

False)

If everything you need is in a self-contained module, or you import the stuff you need manually, just set `autodiscover` to False and don't worry about it :-)

Django

When using Django and the `DJANGO_SETTINGS_MODULE` environment variable is set, the Faust app will scan all packages found in the `INSTALLED_APPS` setting.

If you're using Django you can use this to scan for agents/pages/commands in all packages defined in `INSTALLED_APPS`.

Faust will automatically detect that you're using Django and do the right thing if you do:

```
app = App(..., autodiscover=True)
```

It will find agents and other decorators in all of the reusable Django applications. If you want to manually control what packages are traversed, then provide a list:

```
app = App(..., autodiscover=['package1', 'package2'])
```

or if you want exactly `None` packages to be traversed, then provide a False:

app = App(.., autodiscover=False)

which is the default, so you can simply omit the argument.

Tip: For manual control over autodiscovery, you can also call the *`app.discover()`* method manually.

---

version

type `int`

default 1

Version of the app, that when changed will create a new isolated instance of the application. The first version is 1, the second version is 2, and so on.

Source topics will not be affected by a version change.

Faust applications will use two kinds of topics: source topics, and internally managed topics. The source topics are declared by the producer, and we do not have the opportunity to modify any configuration settings, like number of partitions for a source topic; we may only consume from them. To mark a topic as internal, use: `app.topic(..., internal=True)`.

---

timezone

type `datetime.tzinfo`

default `datetime.timezone.utc`

The timezone used for date-related functionality such as cronjobs.

New in version 1.4.

datadir

type `Union[str, pathlib.Path]`

default `Path(f"{app.conf.id}-data")`

environment `FAUST_DATADIR`, `F_DATADIR`

The directory in which this instance stores the data used by local tables, etc.

See also:

- The data directory can also be set using the *`faust --datadir`* option, from the command-line, so there's usually no reason to provide a default value when creating the app.

tabledir

type `Union[str, pathlib.Path]`

default `"tables"`

The directory in which this instance stores local table data. Usually you will want to configure the *datadir* setting, but if you want to store tables separately you can configure this one.

If the path provided is relative (it has no leading slash), then the path will be considered to be relative to the *datadir* setting.

id_format

type `str`

default `"{id}-v{self.version}"`

The format string used to generate the final *id* value by combining it with the *version* parameter.

logging_config

New in version 1.5.0.

Optional dictionary for logging configuration, as supported by `logging.config.dictConfig()`.

loghandlers

type `List[logging.LogHandler]`

default `[]`

Specify a list of custom log handlers to use in worker instances.

origin

type `str`

default `None`

The reverse path used to find the app, for example if the app is located in:

```
from myproj.app import app
```

Then the `origin` should be `"myproj.app"`.

The **`faust worker`** program will try to automatically set the origin, but if you are having problems with auto generated names then you can set origin manually.

Serialization Settings

key_serializer

type `Union[str, Codec]`

default `"raw"`

Serializer used for keys by default when no serializer is specified, or a model is not being used.

This can be the name of a serializer/codec, or an actual *faust.serializers.codecs.Codec* instance.

See also:

- The *Codecs* section in the model guide – for more information about codecs.

value_serializer

type `Union[str, Codec]`

default `"json"`

Serializer used for values by default when no serializer is specified, or a model is not being used.

This can be string, the name of a serializer/codec, or an actual *faust.serializers.codecs.Codec* instance.

See also:

- The *Codecs* section in the model guide – for more information about codecs.

Topic Settings

topic_replication_factor

type `int`

default `1`

The default replication factor for topics created by the application.

Note: Generally this should be the same as the configured replication factor for your Kafka cluster.

---

topic_partitions

type `int`

default `8`

Default number of partitions for new topics.

Note: This defines the maximum number of workers we could distribute the workload of the application (also sometimes referred as the sharding factor of the application).

---

topic_allow_declare

New in version 1.5.

type `bool`

default `True`

This setting disables the creation of internal topics.

Faust will only create topics that it considers to be fully owned and managed, such as intermediate repartition topics, table changelog topics etc.

Some Kafka managers does not allow services to create topics, in that case you should set this to `False`.

topic_disable_leader

type `bool`

default `False`

This setting disables the creation of the leader election topic.

If you're not using the `on_leader=True` argument to task/timer/etc., decorators then use this setting to disable creation of the topic.

Advanced Broker Settings

broker_consumer

type `Union[str, URL, List[URL]]`

default `None`

You can use this setting to configure the transport used for producing and consuming separately.

If not set the value found in *broker* will be used.

broker_producer

type `Union[str, URL, List[URL]]`

default `None`

You can use this setting to configure the transport used for producing and consuming separately.

If not set the value found in *broker* will be used.

broker_client_id

type `str`

default `f"faust-{VERSION}"`

There is rarely any reason to configure this setting.

The client id is used to identify the software used, and is not usually configured by the user.

broker_request_timeout

New in version 1.4.0.

type `int`

default `90.0` (seconds)

Kafka client request timeout.

Note: The request timeout must not be less than the *broker_session_timeout*.

---

broker_commit_every

type `int`

default `10_000`

Commit offset every n messages.

See also *broker_commit_interval*, which is how frequently we commit on a timer when there are few messages being received.

broker_commit_interval

type `float`, `timedelta`

default `2.8`

How often we commit messages that have been fully processed (*acked*).

broker_commit_livelock_soft_timeout

type `float`, `timedelta`

default `300.0` (five minutes)

How long time it takes before we warn that the Kafka commit offset has not advanced (only when processing messages).

broker_check_crcs

type `bool`

default `True`

Automatically check the CRC32 of the records consumed.

broker_heartbeat_interval

New in version 1.0.11.

type `int`

default `3.0` (three seconds)

How often we send heartbeats to the broker, and also how often we expect to receive heartbeats from the broker.

If any of these time out, you should increase this setting.

broker_session_timeout

New in version 1.0.11.

type `int`

default `60.0` (one minute)

How long to wait for a node to finish rebalancing before the broker will consider it dysfunctional and remove it from the cluster.

Increase this if you experience the cluster being in a state of constantly rebalancing, but make sure you also increase the *broker_heartbeat_interval* at the same time.

Note: The session timeout must not be greater than the *broker_request_timeout*.

---

broker_max_poll_records

New in version 1.4.

type `int`

default `None`

The maximum number of records returned in a single call to poll(). If you find that your application needs more time to process messages you may want to adjust *broker_max_poll_records* to tune the number of records that must be handled on every loop iteration.

broker_max_poll_interval

New in version 1.7.

type `float`

default `1000.0`

The maximum allowed time (in seconds) between calls to consume messages If this interval is exceeded the consumer is considered failed and the group will rebalance in order to reassign the partitions to another consumer group member. If API methods block waiting for messages, that time does not count against this timeout.

See KIP-62 for technical details.

Advanced Consumer Settings

consumer_max_fetch_size

New in version 1.4.

type `int`

default `4*1024**2`

The maximum amount of data per-partition the server will return. This size must be at least as large as the maximum message size.

consumer_auto_offset_reset

New in version 1.5.

type `string`

default `"earliest"`

Where the consumer should start reading messages from when there is no initial offset, or the stored offset no longer exists, e.g. when starting a new consumer for the first time. Options include 'earliest', 'latest', 'none'.

ConsumerScheduler

New in version 1.5.

type `Union[str, Type[SchedulingStrategyT]`

default `faust.transport.utils.DefaultSchedulingStrategy`

A strategy which dictates the priority of topics and partitions for incoming records. The default strategy does first round-robin over topics and then round-robin over partitions.

Example using a class:

```
class MySchedulingStrategy(DefaultSchedulingStrategy):
    ...

app = App(..., ConsumerScheduler=MySchedulingStrategy)
```

Example using the string path to a class:

```
app = App(..., ConsumerScheduler='myproj.MySchedulingStrategy')
```

Advanced Producer Settings

producer_compression_type

type string

default None

The compression type for all data generated by the producer. Valid values are *gzip*, *snappy*, *lz4*, or None.

producer_linger_ms

type int

default 0

Minimum time to batch before sending out messages from the producer.

Should rarely have to change this.

producer_max_batch_size

type int

default 16384

Max size of each producer batch, in bytes.

producer_max_request_size

type int

default 1000000

Maximum size of a request in bytes in the producer.

Should rarely have to change this.

producer_acks

type int

default -1

The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. The following settings are common:

- 0: Producer will not wait for any acknowledgment from the server at all. The message will immediately be considered sent. (Not recommended)
- 1: The broker leader will write the record to its local log but will respond without awaiting full acknowledgment from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost.

- `-1`: The broker leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee.

producer_request_timeout

New in version 1.4.

type `float`, `datetime.timedelta`

default `1200.0` (20 minutes)

Timeout for producer operations. This is set high by default, as this is also the time when producer batches expire and will no longer be retried.

producer_api_version

New in version 1.5.3.

type `str`

default `"auto"`

Negotiate producer protocol version.

The default value - "auto" means use the latest version supported by both client and server.

Any other version set means you are requesting a specific version of the protocol.

Example Kafka uses:

Disable sending headers for all messages produced

Kafka headers support was added in Kafka 0.11, so you can specify `api_version="0.10"` to remove the headers from messages.

producer_partitioner

New in version 1.2.

type `Callable[[bytes, List[int], List[int]], int]`

default `None`

The Kafka producer can be configured with a custom partitioner to change how keys are partitioned when producing to topics.

The default partitioner for Kafka is implemented as follows, and can be used as a template for your own partitioner:

```
import random
from typing import List
from kafka.partitioner.hashed import murmur2

def partition(key: bytes,
              all_partitions: List[int],
              available: List[int]) -> int:
```

(continues on next page)

(continued from previous page)

```
    """Default partitioner.

    Hashes key to partition using murmur2 hashing (from java client)
    If key is None, selects partition randomly from available,
    or from all partitions if none are currently available

    Arguments:
        key: partitioning key
        all_partitions: list of all partitions sorted by partition ID.
        available: list of available partitions in no particular order
    Returns:
        int: one of the values from ``all_partitions`` or ``available``.
    """
    if key is None:
        source = available if available else all_paritions
        return random.choice(source)
    index: int = murmur2(key)
    index &= 0x7fffffff
    index %= len(all_partitions)
    return all_partitions[index]
```

Advanced Table Settings

`table_cleanup_interval`

type `float, timedelta`

default `30.0`

How often we cleanup tables to remove expired entries.

`table_standby_replicas`

type `int`

default `1`

The number of standby replicas for each table.

`table_key_index_size`

New in version 1.8.

type `int`

default `1000`

Tables keep a cache of key to partition number to speed up table lookups.

This setting configures the maximum size of that cache.

Advanced Stream Settings

stream_buffer_maxsize

type int

default 4096

This setting control back pressure to streams and agents reading from streams.

If set to 4096 (default) this means that an agent can only keep at most 4096 unprocessed items in the stream buffer.

Essentially this will limit the number of messages a stream can "prefetch".

Higher numbers gives better throughput, but do note that if your agent sends messages or update tables (which sends changelog messages).

This means that if the buffer size is large, the *broker_commit_interval* or *broker_commit_every* settings must be set to commit frequently, avoiding back pressure from building up.

A buffer size of 131_072 may let you process over 30,000 events a second as a baseline, but be careful with a buffer size that large when you also send messages or update tables.

stream_recovery_delay

type Union[float, datetime.timedelta]

default 0.0

Number of seconds to sleep before continuing after rebalance. We wait for a bit to allow for more nodes to join/leave before starting recovery tables and then processing streams. This to minimize the chance of errors rebalancing loops.

Changed in version 1.5.3: Disabled by default.

stream_wait_empty

type bool

default True

This setting controls whether the worker should wait for the currently processing task in an agent to complete before rebalancing or shutting down.

On rebalance/shut down we clear the stream buffers. Those events will be reprocessed after the rebalance anyway, but we may have already started processing one event in every agent, and if we rebalance we will process that event again.

By default we will wait for the currently active tasks, but if your streams are idempotent you can disable it using this setting.

stream_publish_on_commit

type bool

default False

If enabled we buffer up sending messages until the source topic offset related to that processing is committed. This means when we do commit, we may have buffered up a LOT of messages so commit needs to happen frequently (make sure to decrease *broker_commit_every*).

Advanced Worker Settings

worker_redirect_stdouts

type bool

default True

Enable to have the worker redirect output to sys.stdout and sys.stderr to the Python logging system.

Enabled by default.

worker_redirect_stdouts_level

type str/int

default "WARN"

The logging level to use when redirect STDOUT/STDERR to logging.

Advanced Web Server Settings

web

New in version 1.2.

type str

default URL("aiohttp://")

The web driver to use.

web_enabled

New in version 1.2.

type bool

default True

Enable web server and other web components.

This option can also be set using *faust worker --without-web*.

web_transport

New in version 1.2.

type `str`

default `URL("tcp://")`

The network transport used for the web server.

Default is to use TCP, but this setting also enables you to use Unix domain sockets. To use domain sockets specify an URL including the path to the file you want to create like this:

```
unix:///tmp/server.sock
```

This will create a new domain socket available in `/tmp/server.sock`.

canonical_url

type `str`

default `URL(f"http://{web_host}:{web_port}")`

You shouldn't have to set this manually.

The canonical URL defines how to reach the web server on a running worker node, and is usually set by combining the *`faust worker --web-host`* and *`faust worker --web-port`* command line arguments, not by passing it as a keyword argument to `App`.

web_host

New in version 1.2.

type `str`

default `f"{socket.gethostname()}"`

Hostname used to access this web server, used for generating the *`canonical_url`* setting.

This option is usually set by *`faust worker --web-host`*, not by passing it as a keyword argument to `app`.

web_port

New in version 1.2.

type `int`

default `6066`

A port number between 1024 and 65535 to use for the web server.

This option is usually set by *`faust worker --web-port`*, not by passing it as a keyword argument to `app`.

web_bind

New in version 1.2.

type `str`

default `"0.0.0.0"`

The IP network address mask that decides what interfaces the web server will bind to.

By default this will bind to all interfaces.

This option is usually set by *`faust worker --web-bind`*, not by passing it as a keyword argument to `app`.

web_in_thread

New in version 1.5.

type `bool`

default `False`

Run the web server in a separate thread.

Use this if you have a large value for *`stream_buffer_maxsize`* and want the web server to be responsive when the worker is otherwise busy processing streams.

Note: Running the web server in a separate thread means web views and agents will not share the same event loop.

---

web_cors_options

New in version 1.5.

type `Mapping[str, ResourceOptions]`

default `None`

Enable Cross-Origin Resource Sharing options for all web views in the internal web server.

This should be specified as a dictionary of URLs to `ResourceOptions`:

```
app = App(..., web_cors_options={
    'http://foo.example.com': ResourceOptions(
        allow_credentials=True,
        allow_methods='*',
    )
})
```

Individual views may override the CORS options used as arguments to to `@app.page` and `blueprint.route`.

See also:

aiohttp_cors: https://github.com/aio-libs/aiohttp-cors

Advanced Agent Settings

agent_supervisor

type str:/mode.SupervisorStrategyT

default mode.OneForOneSupervisor

An agent may start multiple instances (actors) when the concurrency setting is higher than one (e.g. `@app.agent(concurrency=2)`).

Multiple instances of the same agent are considered to be in the same supervisor group.

The default supervisor is the mode.OneForOneSupervisor: if an instance in the group crashes, we restart that instance only.

These are the supervisors supported:

- mode.OneForOneSupervisor

  If an instance in the group crashes we restart only that instance.

- mode.OneForAllSupervisor

  If an instance in the group crashes we restart the whole group.

- mode.CrashingSupervisor

  If an instance in the group crashes we stop the whole application, and exit so that the Operating System supervisor can restart us.

- mode.ForfeitOneForOneSupervisor

  If an instance in the group crashes we give up on that instance and never restart it again (until the program is restarted).

- mode.ForfeitOneForAllSupervisor

  If an instance in the group crashes we stop all instances in the group and never restarted them again (until the program is restarted).

Agent RPC Settings

reply_to

type str

default *str(uuid.uuid4())*

The name of the reply topic used by this instance. If not set one will be automatically generated when the app is created.

reply_create_topic

type `bool`

default `False`

Set this to `True` if you plan on using the RPC with agents.

This will create the internal topic used for RPC replies on that instance at startup.

reply_expires

type `Union[float, datetime.timedelta]`

default `timedelta(days=1)`

The expiry time (in seconds `float`, or `timedelta`), for how long replies will stay in the instances local reply topic before being removed.

reply_to_prefix

type `str`

default `"f-reply-"`

The prefix used when generating reply topic names.

Extension Settings

Agent

type `Union[str, Type]`

default *faust.Agent*

The *Agent* class to use for agents, or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
class MyAgent(faust.Agent):
    ...

app = App(..., Agent=MyAgent)
```

Example using the string path to a class:

```
app = App(..., Agent='myproj.agents.Agent')
```

Event

type `Union[str, Type]`

default *faust.Event*

The *Event* class to use for creating new event objects, or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
class MyBaseEvent(faust.Event):
    ...

app = App(..., Event=MyBaseEvent)
```

Example using the string path to a class:

```
app = App(..., Event='myproj.events.Event')
```

Schema

type `Union[str, Type]`

default *faust.Schema*

The *Schema* class to use as the default schema type when no schema specified. or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
class MyBaseSchema(faust.Schema):
    ...

app = App(..., Schema=MyBaseSchema)
```

Example using the string path to a class:

```
app = App(..., Schema='myproj.schemas.Schema')
```

Stream

type `Union[str, Type]`

default *faust.Stream*

The *Stream* class to use for streams, or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
class MyBaseStream(faust.Stream):
    ...

app = App(..., Stream=MyBaseStream)
```

Example using the string path to a class:

```
app = App(..., Stream='myproj.streams.Stream')
```

Table

type `Union[str, Type[TableT]]`

default *faust.Table*

The *Table* class to use for tables, or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
class MyBaseTable(faust.Table):
    ...

app = App(..., Table=MyBaseTable)
```

Example using the string path to a class:

```
app = App(..., Table='myproj.tables.Table')
```

SetTable

type `Union[str, Type[TableT]]`

default *faust.SetTable*

The *SetTable* class to use for table-of-set tables, or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
class MySetTable(faust.SetTable):
    ...

app = App(..., Table=MySetTable)
```

Example using the string path to a class:

```
app = App(..., Table='myproj.tables.MySetTable')
```

GlobalTable

type `Union[str, Type[GlobalTableT]]`

default *faust.GlobalTable*

The *GlobalTable* class to use for tables, or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
class MyBaseGlobalTable(faust.GlobalTable):
    ...

app = App(..., GlobalTable=MyBaseGlobalTable)
```

Example using the string path to a class:

```
app = App(..., GlobalTable='myproj.tables.GlobalTable')
```

SetGlobalTable

type `Union[str, Type[GlobalTableT]]`

default *faust.SetGlobalTable*

The *SetGlobalTable* class to use for tables, or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
class MyBaseSetGlobalTable(faust.SetGlobalTable):
    ...

app = App(..., SetGlobalTable=MyBaseGlobalSetTable)
```

Example using the string path to a class:

```
app = App(..., SetGlobalTable='myproj.tables.SetGlobalTable')
```

TableManager

type `Union[str, Type[TableManagerT]]`

default *faust.tables.TableManager*

The *TableManager* used for managing tables, or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
from faust.tables import TableManager

class MyTableManager(TableManager):
    ...

app = App(..., TableManager=MyTableManager)
```

Example using the string path to a class:

```
app = App(..., TableManager='myproj.tables.TableManager')
```

Serializers

type `Union[str, Type[RegistryT]]`

default `faust.serializers.Registry`

The `Registry` class used for serializing/deserializing messages; or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
from faust.serialiers import Registry

class MyRegistry(Registry):
    ...

app = App(..., Serializers=MyRegistry)
```

Example using the string path to a class:

```
app = App(..., Serializers='myproj.serializers.Registry')
```

Worker

type `Union[str, Type[WorkerT]]`

default *faust.Worker*

The *Worker* class used for starting a worker for this app; or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
import faust

class MyWorker(faust.Worker):
    ...

app = faust.App(..., Worker=Worker)
```

Example using the string path to a class:

```
app = faust.App(..., Worker='myproj.workers.Worker')
```

PartitionAssignor

type `Union[str, Type[PartitionAssignorT]]`

default `faust.assignor.PartitionAssignor`

The `PartitionAssignor` class used for assigning topic partitions to worker instances; or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
from faust.assignor import PartitionAssignor

class MyPartitionAssignor(PartitionAssignor):
    ...

app = App(..., PartitionAssignor=PartitionAssignor)
```

Example using the string path to a class:

```
app = App(..., Worker='myproj.assignor.PartitionAssignor')
```

LeaderAssignor

type `Union[str, Type[LeaderAssignorT]]`

default `faust.assignor.LeaderAssignor`

The `LeaderAssignor` class used for assigning a master Faust instance for the app; or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
from faust.assignor import LeaderAssignor

class MyLeaderAssignor(LeaderAssignor):
    ...

app = App(..., LeaderAssignor=LeaderAssignor)
```

Example using the string path to a class:

```
app = App(..., Worker='myproj.assignor.LeaderAssignor')
```

Router

type `Union[str, Type[RouterT]]`

default *faust.app.router.Router*

The `Router` class used for routing requests to a worker instance having the partition for a specific key (e.g. table key); or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
from faust.router import Router

class MyRouter(Router):
    ...

app = App(..., Router=Router)
```

Example using the string path to a class:

```
app = App(..., Router='myproj.routers.Router')
```

Topic

type `Union[str, Type[TopicT]]`

default *faust.Topic*

The *Topic* class used for defining new topics; or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
import faust

class MyTopic(faust.Topic):
```

(continues on next page)

(continued from previous page)

```
    ...

app = faust.App(..., Topic=MyTopic)
```

Example using the string path to a class:

```
app = faust.App(..., Topic='myproj.topics.Topic')
```

HttpClient

type `Union[str, Type[HttpClientT]]`

default `aiohttp.client.ClientSession`

The `aiohttp.client.ClientSession` class used as a HTTP client; or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
import faust
from aiohttp.client import ClientSession

class HttpClient(ClientSession):
    ...

app = faust.App(..., HttpClient=HttpClient)
```

Example using the string path to a class:

```
app = faust.App(..., HttpClient='myproj.http.HttpClient')
```

Monitor

type `Union[str, Type[SensorT]]`

default *faust.sensors.Monitor*

The *Monitor* class as the main sensor gathering statistics for the application; or the fully-qualified path to one (supported by `symbol_by_name()`).

Example using a class:

```
import faust
from faust.sensors import Monitor

class MyMonitor(Monitor):
    ...

app = faust.App(..., Monitor=MyMonitor)
```

Example using the string path to a class:

```
app = faust.App(..., Monitor='myproj.monitors.Monitor')
```

1.4.13 Installation

Installation

You can install Faust either via the Python Package Index (PyPI) or from source.

To install using *pip*:

```
$ pip install -U faust
```

Bundles

Faust also defines a group of setuptools extensions that can be used to install Faust and the dependencies for a given feature.

You can specify these in your requirements or on the **pip** command-line by using brackets. Separate multiple bundles using the comma:

```
$ pip install "faust[rocksdb]"

$ pip install "faust[rocksdb,uvloop,fast,redis]"
```

The following bundles are available:

Stores

faust[rocksdb] for using RocksDB for storing Faust table state.

Recommended in production.

Caching

faust[redis] for using *Redis*_ as a simple caching backend (Memcached-style).

Codecs

faust[yaml] for using YAML and the PyYAML library in streams.

Optimization

faust[fast] for installing all the available C speedup extensions to Faust core.

Sensors

faust[datadog] for using the Datadog Faust monitor.

faust[statsd] for using the Statsd Faust monitor.

Event Loops

faust[uvloop] for using Faust with uvloop.

faust[eventlet] for using Faust with eventlet

Debugging

faust[debug] for using aiomonitor to connect and debug a running Faust worker.

faust[setproctitle] when the setproctitle module is installed the Faust worker will use it to set a nicer process name in **ps**/**top** listings. Also installed with the `fast` and `debug` bundles.

Downloading and installing from source

Download the latest version of Faust from http://pypi.org/project/faust

You can install it by doing:

```
$ tar xvfz faust-0.0.0.tar.gz
$ cd faust-0.0.0
$ python setup.py build
# python setup.py install
```

The last command must be executed as a privileged user if you are not currently using a virtualenv.

Using the development version

With pip

You can install the latest snapshot of Faust using the following **pip** command:

```
$ pip install https://github.com/robinhood/faust/zipball/master#egg=faust
```

1.4.14 Kafka - The basics you need to know

Kafka is a distributed streaming platform which uses logs as the unit of storage for messages passed within the system. It is horizontally scalable, fault-tolerant, fast, and runs in production in thousands of companies. Likely your business is already using it in some form.

What you must know about Apache Kafka to use Faust

Topics

A topic is a stream name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it. Topics are also the base abstraction of where data lives within Kafka. Each topic is backed by logs which are *partitioned* and distributed.

Faust uses the abstraction of a topic to both consume data from a stream as well as publish data to more streams represented by Kafka topics. A Faust application needs to be consuming from at least one topic and may create many intermediate topics as a by-product of your streaming application. Every topic that is not created by Faust internally can be thought of as a source (for your application to process) or sink (for other systems to pick up).

Partitions

Partitions are the fundamental unit within Kafka where data lives. Every topic is split into one or more partitions. These partitions are represented internally as logs and messages always make their way to one partition (log). Each partition is replicated and has one leader at any point in time.

Faust uses the notion of a partition to maintain order amongst messages and as a way to split the processing of data to increase throughput. A Faust application uses the notion of a "key" to make sure that messages that should appear together and be processed on the same box do end up on the same box.

Fault Tolerance

Every partition is replicated with the total copies represented by the In Sync Replicas (ISR). Every ISR is a candidate to take over as the new leader should the current leader fail. The maximum number of faulty Kafka brokers that can be tolerated is the number of ISR - 1. I.e., if every partition has three replicas, all fault tolerance guarantees hold as long as at least one replica is functional

Faust has the same guarantees that Kafka offers with regards to fault tolerance of the data.

Distribution of load/work

For every partition, all reads and writes are routed to the leader of that partition. For a specific topic, the load is as distributed as the number of partitions. *Note:* Since the partition is the lowest degree of parallel processing of messages, the number of partitions control how much many parallel instances of the consumers can operate on messages.

Faust uses parallel consumers and therefore is also limited by the number of partitions to dictate how many concurrent Faust application instances can run to distribute work. Extra Faust application instances beyond the source topic partition count will be idle and not improve message processing rates.

Offsets For every <topic, partition> combination Kafka keeps track of the offset of messages in the log to know where new messages should be appended. On a consumer level, offsets are maintained at the <group, topic, partition> level for consumers to know where to continue consuming for a given "group". The group acts as a namespace for consumers to register when multiple consumers want to share the load on a single topic.

Kafka maintains processing guarantees of at least once by committing offsets after message consumption. Once an offset has been committed at the consumer level, the message at that offset for the <group, topic, partition> will not be reread.

Faust uses the notion of a group to maintain a namespace within an app. Faust commits offsets after when a message is processed through all of its operations. Faust allows a configurable *commit interval* which makes sure that all messages that have been processed completely since the last interval will be committed.

Log Compaction

Log compaction is a methodology Kafka uses to make sure that as data for a key changes it will not affect the size of the log such that every state change is maintained for all time. Only the most recent value is guaranteed to be available. Periodic compaction removes all values for a key except the last one.

Tables in Faust use log compaction to ensure table state can be recovered without a large space overhead.

This summary and information about Kafka is adapted from original documentation on Kafka available at https://kafka.apache.org/

1.4.15 Debugging

Debugging with aiomonitor

To use the debugging console you first need to install the aiomonitor library:

```
$ pip install aiomonitor
```

You can also install it as part of a *bundle*:

```
$ pip install -U faust[debug]
```

After aiomonitor is installed you may start the worker with the `--debug` option enabled:

```
$ faust -A myapp --debug worker -l info
┌faust v0.9.20──────────────────────────────────┐
| id          | word-counts                     |
| transport   | kafka://localhost:9092          |
| store       | rocksdb:                        |
| web         | http://localhost:6066/          |
| log         | -stderr- (info)                 |
| pid         | 55522                           |
| hostname    | grainstate.local                |
| platform    | CPython 3.6.3 (Darwin x86_64)   |
| drivers     | aiokafka=0.3.2 aiohttp=2.3.7    |
| datadir     | /opt/devel/faust/word-counts-data |
└─────────────┴─────────────────────────────────┘
[2018-01-04 12:41:07,635: INFO]: Starting aiomonitor at 127.0.0.1:50101
[2018-01-04 12:41:07,637: INFO]: Starting console at 127.0.0.1:50101
[2018-01-04 12:41:07,638: INFO]: [^Worker]: Starting...
[2018-03-13 13:41:39,275: INFO]: [^-App]: Starting...
[2018-01-04 12:41:07,638: INFO]: [^--Web]: Starting...
[...]
```

From the log output you can tell that the aiomonitor console was started on the local port 50101. If you get a different output, such as that the port is already taken you can set a custom port using the `--console-port`.

Once you have the port number, you can telnet into the console to use it:

```
$ telnet localhost 50101
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Asyncio Monitor: 38 tasks running
Type help for commands

monitor >>>
```

Type `help` and then press enter to see a list of available commands:

```
monitor >>> help
Commands:
         ps               : Show task table
         where taskid     : Show stack frames for a task
         cancel taskid    : Cancel an indicated task
         signal signame   : Send a Unix signal
         console          : Switch to async Python REPL
         quit             : Leave the monitor
monitor >>>
```

To exit out of the console you can either type *quit* at the `monitor >>` prompt. If that is unresponsive you may hit the special telnet escape character (`Ctrl-]`), to drop you into the telnet command console, and from there you just type *quit* to exit out of the telnet session:

```
$> telnet localhost 50101
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Asyncio Monitor: 38 tasks running
Type help for commands
monitor >>> ^]
telnet> quit
Connection closed.
```

1.4.16 Workers Guide

Managing individual instances

This part describes managing individual instances and is more relevant in development.

Make sure you also read the ref:*worker-cluster* section of this guide for production deployments.

Starting a worker

Daemonizing

You probably want to use a daemonization tool to start the worker in the background. Use *systemd*, *supervisord* or any of the tools you usually use to start services. We hope to have a detailed guide for each of these soon.

If you have defined a Faust app in the module `proj.py`:

```
# proj.py
import faust

app = faust.App('proj', broker='kafka://localhost:9092')

@app.agent()
async def process(stream):
    async for value in stream:
        print(value)
```

You can start the worker in the foreground by executing the command:

```
$ faust -A proj worker -l info
```

For a full list of available command-line options simply do:

```
$ faust worker --help
```

You can start multiple workers for the same app on the same machine, but be sure to provide a unique web server port to each worker, and also a unique data directory.

Start first worker:

```
$ faust --datadir=/var/faust/worker1 -A proj -l info worker --web-port=6066
```

Then start the second worker:

```
$ faust --datadir=/var/faust/worker2 -A proj -l info worker --web-port=6067
```

Sharing Data Directories

Worker instances should not share data directories, so make sure to specify a different data directory for every worker instance.

Stopping a worker

Shutdown is accomplished using the `TERM` signal.

When shutdown is initiated the worker will finish all currently executing tasks before it actually terminates. If these tasks are important, you should wait for it to finish before doing anything drastic, like sending the `KILL` signal.

If the worker won't shutdown after considerate time, for being stuck in an infinite-loop or similar, you can use the `KILL` signal to force terminate the worker. The tasks that did not complete will be executed again by another worker.

Starting subprocesses

For Faust applications that start subprocesses as a side effect of processing the stream, you should know that the "double-fork" problem on Unix means that the worker will not be able to reap its children when killed using the `KILL` signal.

To kill the worker and any child processes, this command usually does the trick:

```
$ pkill -9 -f 'faust'
```

If you don't have the **`pkill`** command on your system, you can use the slightly longer version:

```
$ ps auxww | grep 'faust' | awk '{print $2}' | xargs kill -9
```

Restarting a worker

To restart the worker you should send the *TERM* signal and start a new instance.

Kafka Rebalancing

When using Kafka, stopping or starting new workers will trigger a rebalancing operation that require all workers to stop stream processing.

See *Managing a cluster* for more information.

Process Signals

The worker's main process overrides the following signals:

`TERM`	Warm shutdown, wait for tasks to complete.
`QUIT`	Cold shutdown, terminate ASAP
`USR1`	Dump traceback for all active threads in logs

Managing a cluster

In production deployments the management of a cluster of worker instances is complicated by the Kafka rebalancing strategy.

Every time a new worker instance joins or leaves, the Kafka broker will ask all instances to perform a "rebalance" of available partitions.

This "stop the world" process will temporarily halt processing of all streams, and if this rebalancing operation is not managed properly, you may end up in a state of perpetual rebalancing: the workers will continually trigger rebalances to occur, effectively halting processing of the stream.

Note: The Faust web server is not affected by rebalancing, and will still serve web requests.

This is important to consider when using tables and serving table data over HTTP. Tables exposed in this manner will be eventually consistent and may serve stale data during a rebalancing operation.

When will rebalancing occur? It will occur should you restart one of the workers, or when restarting workers to deploy changes, and also if you change the number of partitions for a topic to scale a cluster up or down.

Restarting a cluster

To minimize the chance of rebalancing problems we suggest you use the following strategy to restart all the workers:

1) Stop 50% of the workers (and wait for them to shut down).
2) Start the workers you just stopped and wait for them to fully start.
3) Stop the other half of the workers (and wait for them to shut down).
4) Start the other half of the workers.

This should both minimize rebalancing issues and also keep the built-in web servers up and available to serve HTTP requests.

KIP-441 and the future…

The Kafka developer community have proposed a solution to this problem, so in the future we may have an easier way to deploy code changes and even support autoscaling of workers.

See KIP-441: Smooth Scaling Out for Kafka Streams for more information.

1.5 Frequently Asked Questions (FAQ)

1.5.1 FAQ

Can I use Faust with Django/Flask/etc.?

Yes! Use eventlet as a bridge to integrate with `asyncio`.

Using eventlet

This approach works with any blocking Python library that can work with eventlet.

Using eventlet requires you to install the aioeventlet module, and you can install this as a bundle along with Faust:

```
$ pip install -U faust[eventlet]
```

Then to actually use eventlet as the event loop you have to either use the `-L` argument to the **`faust`** program:

```
$ faust -L eventlet -A myproj worker -l info
```

or add `import mode.loop.eventlet` at the top of your entry point script:

```
#!/usr/bin/env python3
import mode.loop.eventlet  # noqa
```

Warning: It's very important this is at the very top of the module, and that it executes before you import libraries.

Can I use Faust with Tornado?

Yes! Use the `tornado.platform.asyncio` bridge: http://www.tornadoweb.org/en/stable/asyncio.html

Can I use Faust with Twisted?

Yes! Use the `asyncio` reactor implementation: https://twistedmatrix.com/documents/17.1.0/api/twisted.internet.asyncioreactor.html

Will you support Python 3.5 or earlier?

There are no immediate plans to support Python 3.5, but you are welcome to contribute to the project.

Here are some of the steps required to accomplish this:

- Source code transformation to rewrite variable annotations to comments

  for example, the code:

  ```
      class Point:
          x: int = 0
          y: int = 0

  must be rewritten into::

      class Point:
          x = 0  # type: int
          y = 0  # type: int
  ```

- Source code transformation to rewrite async functions

  for example, the code:

  ```
  async def foo():
      await asyncio.sleep(1.0)
  ```

must be rewritten into:

```
@coroutine
def foo():
    yield from asyncio.sleep(1.0)
```

Will you support Python 2?

There are no plans to support Python 2, but you are welcome to contribute to the project (details in the question above is relevant also for Python 2).

I get a maximum number of open files exceeded error by RocksDB when running a Faust app locally. How can I fix this?

You may need to increase the limit for the maximum number of open files. The following post explains how to do so on OS X: https://blog.dekstroza.io/ulimit-shenanigans-on-osx-el-capitan/

What kafka versions faust supports?

Faust supports kafka with version >= 0.10.

1.6 API Reference

Release 1.9

Date Jan 09, 2020

1.6.1 Faust

faust

Python Stream processing.

class faust.**Service**(*, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None

An asyncio service that can be started/stopped/restarted.

Keyword Arguments

- **beacon** (*NodeT*) – Beacon used to track services in a graph.
- **loop** (*asyncio.AbstractEventLoop*) – Event loop object.

abstract = False

class Diag(*service: mode.types.services.ServiceT*) → None

Service diagnostics.

This can be used to track what your service is doing. For example if your service is a Kafka consumer with a background thread that commits the offset every 30 seconds, you may want to see when this happens:

```
DIAG_COMMITTING = 'committing'

class Consumer(Service):

    @Service.task
    async def _background_commit(self) -> None:
        while not self.should_stop:
            await self.sleep(30.0)
            self.diag.set_flag(DIAG_COMITTING)
            try:
                await self._consumer.commit()
            finally:
                self.diag.unset_flag(DIAG_COMMITTING)
```

The above code is setting the flag manually, but you can also use a decorator to accomplish the same thing:

```
@Service.timer(30.0)
async def _background_commit(self) -> None:
    await self.commit()

@Service.transitions_with(DIAG_COMITTING)
async def commit(self) -> None:
    await self._consumer.commit()
```

set_flag(*flag: str*) → None

Return type None

unset_flag(*flag: str*) → None

Return type None

wait_for_shutdown = False

Set to True if .stop must wait for the shutdown flag to be set.

shutdown_timeout = 60.0

Time to wait for shutdown flag set before we give up.

restart_count = 0

Current number of times this service instance has been restarted.

mundane_level = 'info'

The log level for mundane info such as *starting*, *stopping*, etc. Set this to "debug" for less information.

classmethod from_awaitable(*coro: Awaitable*, ***, *name: str = None*, ***kwargs: Any*) → mode.types.services.ServiceT

Return type ServiceT[]

classmethod task(*fun: Callable[Any, Awaitable[None]]*) → mode.services.ServiceTask

Decorate function to be used as background task.

Example

```
>>> class S(Service):
...
...     @Service.task
...     async def background_task(self):
...         while not self.should_stop:
...             await self.sleep(1.0)
...             print('Waking up')
```

Return type ServiceTask

classmethod timer(*interval: Union[datetime.timedelta, float, str]*) → Callable[Callable, mode.services.ServiceTask]

Background timer executing every n seconds.

Example

```
>>> class S(Service):
...
...     @Service.timer(1.0)
...     async def background_timer(self):
...         print('Waking up')
```

Return type Callable[[Callable], ServiceTask]

classmethod transitions_to(*flag: str*) → Callable

Decorate function to set and reset diagnostic flag.

Return type Callable

async transition_with(*flag: str*, *fut: Awaitable*, **args: Any*, ***kwargs: Any*) → Any

Return type Any

add_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

Add dependency to other service.

The service will be started/stopped with this service.

Return type ServiceT[]

async add_runtime_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

Return type ServiceT[]

async remove_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

Stop and remove dependency of this service.

Return type ServiceT[]

async add_async_context(*context: AsyncContextManager*) → Any

Return type Any

add_context(*context: ContextManager*) → Any

Return type Any

add_future(*coro: Awaitable*) → _asyncio.Future
Add relationship to asyncio.Future.

The future will be joined when this service is stopped.

Return type Future

on_init() → None

Return type None

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of service dependencies for this service.

Return type Iterable[ServiceT[]]

async join_services(*services: Sequence[mode.types.services.ServiceT]*) → None

Return type None

async sleep(*n: Union[datetime.timedelta, float, str], *, loop: asyncio.events.AbstractEventLoop = None*) → None
Sleep for n seconds, or until service stopped.

Return type None

async wait_for_stopped(**coros: Union[Generator[[Any, None], Any], Awaitable, asyncio.locks.Event, mode.utils.locks.Event], timeout: Union[datetime.timedelta, float, str] = None*) → bool

Return type bool

async wait(**coros: Union[Generator[[Any, None], Any], Awaitable, asyncio.locks.Event, mode.utils.locks.Event], timeout: Union[datetime.timedelta, float, str] = None*) → mode.services.WaitResult
Wait for coroutines to complete, or until the service stops.

Return type WaitResult

async wait_many(*coros: Iterable[Union[Generator[[Any, None], Any], Awaitable, asyncio.locks.Event, mode.utils.locks.Event]], *, timeout: Union[datetime.timedelta, float, str] = None*) → mode.services.WaitResult

Return type WaitResult

async wait_first(**coros: Union[Generator[[Any, None], Any], Awaitable, asyncio.locks.Event, mode.utils.locks.Event], timeout: Union[datetime.timedelta, float, str] = None*) → mode.services.WaitResults

Return type WaitResults

async start() → None

Return type None

async maybe_start() → None
Start the service, if it has not already been started.

Return type None

async crash(*reason: BaseException*) → None
Crash the service and all child services.

Return type None

async stop() → None
Stop the service.

Return type None

async restart() → None
Restart this service.

Return type None

service_reset() → None

Return type None

async wait_until_stopped() → None
Wait until the service is signalled to stop.

Return type None

set_shutdown() → None
Set the shutdown signal.

Notes

If *wait_for_shutdown* is set, stopping the service will wait for this flag to be set.

Return type None

itertimer(*interval: Union[datetime.timedelta, float, str], *, max_drift_correction: float = 0.1, loop: asyncio.events.AbstractEventLoop = None, sleep: Callable[..., Awaitable] = None, clock: Callable[float] = <built-in function perf_counter>, name: str = ''*) → AsyncIterator[float]
Sleep `interval` seconds for every iteration.

This is an async iterator that takes advantage of `timer_intervals()` to act as a timer that stop drift from occurring, and adds a tiny amount of drift to timers so that they don't start at the same time.

Uses `Service.sleep` which will bail-out-quick if the service is stopped.

Note: Will sleep the full *interval* seconds before returning from first iteration.

Examples

```
>>> async for sleep_time in self.itertimer(1.0):
...   print('another second passed, just woke up...')
...   await perform_some_http_request()
```

Return type AsyncIterator[float]

property started
Return `True` if the service was started. :rtype: `bool`

property crashed

Return type bool

property should_stop
Return `True` if the service must stop. :rtype: `bool`

property state
Service state - as a human readable string. :rtype: `str`

property label
Label used for graphs. :rtype: str

property shortlabel
Label used for logging. :rtype: str

property beacon
Beacon used to track services in a dependency graph. :rtype: NodeT[~_T]

logger = <Logger mode.services (WARNING)>

property crash_reason

Return type Optional[BaseException]

class faust.**ServiceT**(*, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None
Abstract type for an asynchronous service that can be started/stopped.

See also:

mode.Service.

wait_for_shutdown = False

restart_count = 0

supervisor = None

abstract add_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

Return type ServiceT[]

abstract async add_runtime_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

Return type ServiceT[]

abstract async add_async_context(*context: AsyncContextManager*) → Any

Return type Any

abstract add_context(*context: ContextManager*) → Any

Return type Any

abstract async start() → None

Return type None

abstract async maybe_start() → None

Return type None

abstract async crash(*reason: BaseException*) → None

Return type None

abstract async stop() → None

Return type None

abstract service_reset() → None

Return type None

abstract async restart() → None

Return type None

abstract async wait_until_stopped() → None

Return type None

abstract set_shutdown() → None

Return type None

abstract property started

Return type bool

abstract property crashed

Return type bool

abstract property should_stop

Return type bool

abstract property state

Return type str

abstract property label

Return type str

abstract property shortlabel

Return type str

property beacon

Return type NodeT[~_T]

abstract property loop

Return type AbstractEventLoop

abstract property crash_reason

Return type Optional[BaseException]

class faust.**Agent**(*fun: Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], *, app: faust.types.app.AppT, name: str = None, channel: Union[str, faust.types.channels.ChannelT] = None, concurrency: int = 1, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, on_error: Callable[[AgentT, BaseException], Awaitable] = None, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, help: str = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, isolated_partitions: bool = False, use_reply_headers: bool = None, **kwargs: Any*) → None

Agent.

This is the type of object returned by the `@app.agent` decorator.

supervisor = None

on_init_dependencies() → Iterable[mode.types.services.ServiceT]

Return list of services dependencies required to start agent.

Return type Iterable[ServiceT[]]

async on_start() → None

Call when an agent starts.

Return type None

async on_stop() → None
Call when an agent stops.

Return type None

cancel() → None
Cancel agent and its actor instances running in this process.

Return type None

async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when partitions are revoked.

Return type None

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when partitions are assigned.

Return type None

async on_isolated_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when isolated partitions are revoked.

Return type None

async on_isolated_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when isolated partitions are assigned.

Return type None

async on_shared_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when non-isolated partitions are revoked.

Return type None

async on_shared_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when non-isolated partitions are assigned.

Return type None

info() → Mapping
Return agent attributes as a dictionary.

Return type Mapping[~KT, +VT_co]

clone(*, *cls: Type[faust.types.agents.AgentT] = None*, ***kwargs: Any*) → faust.types.agents.AgentT
Create clone of this agent object.

Keyword arguments can be passed to override any argument supported by *Agent.__init__*.

Return type *AgentT*[]

test_context(*channel: faust.types.channels.ChannelT = None*, *supervisor_strategy: mode.types.supervisors.SupervisorStrategyT = None*, *on_error: Callable[[AgentT, BaseException], Awaitable] = None*, ***kwargs: Any*) → faust.types.agents.AgentTestWrapperT
Create new unit-testing wrapper for this agent.

Return type *AgentTestWrapperT*[]

actor_from_stream(*stream: Optional[faust.types.streams.StreamT], *, index: int = None, active_partitions: Set[faust.types.tuples.TP] = None, channel: faust.types.channels.ChannelT = None*) → faust.types.agents.ActorT[Union[AsyncIterable, Awaitable]]

Create new actor from stream.

Return type *ActorT*[]

add_sink(*sink: Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]*) → None

Add new sink to further handle results from this agent.

Return type None

stream(*channel: faust.types.channels.ChannelT = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, ***kwargs: Any*) → faust.types.streams.StreamT

Create underlying stream used by this agent.

Return type *StreamT*[+T_co]

async cast(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*) → None

RPC operation: like *ask()* but do not expect reply.

Cast here is like "casting a spell", and will not expect a reply back from the agent.

Return type None

async ask(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None*) → Any

RPC operation: ask agent for result of processing value.

This version will wait until the result is available and return the processed value.

Return type Any

async ask_nowait(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None, force: bool = False*) → faust.agents.replies.ReplyPromise

RPC operation: ask agent for result of processing value.

This version does not wait for the result to arrive, but instead returns a promise of future evaluation.

Return type *ReplyPromise*

async send(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Send message to topic used by agent.

Return type Awaitable[*RecordMetadata*]

map(*values: Union[AsyncIterable, Iterable], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator
RPC map operation on a list of values.

A map operation iterates over results as they arrive. See *join()* and *kvjoin()* if you want them in order.

Return type AsyncIterator[+T_co]

kvmap(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator[str]
RPC map operation on a list of `(key, value)` pairs.

A map operation iterates over results as they arrive. See *join()* and *kvjoin()* if you want them in order.

Return type AsyncIterator[str]

async join(*values: Union[AsyncIterable[Union[bytes, faust.types.core._ModelT, Any]], Iterable[Union[bytes, faust.types.core._ModelT, Any]]], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]
RPC map operation on a list of values.

A join returns the results in order, and only returns once all values have been processed.

Return type List[Any]

async kvjoin(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]
RPC map operation on list of `(key, value)` pairs.

A join returns the results in order, and only returns once all values have been processed.

Return type List[Any]

get_topic_names() → Iterable[str]
Return list of topic names this agent subscribes to.

Return type Iterable[str]

property channel
Return channel used by agent. :rtype: *ChannelT*[]

property channel_iterator
Return channel agent iterates over. :rtype: AsyncIterator[+T_co]

property label
Return human-readable description of agent. :rtype: str

property shortlabel
Return short description of agent. :rtype: str

logger = <Logger faust.agents.agent (WARNING)>

class faust.**App**(*id: str, *, monitor: faust.sensors.monitor.Monitor = None, config_source: Any = None, loop: asyncio.events.AbstractEventLoop = None, beacon: mode.utils.types.trees.NodeT = None, **options: Any*) → None
Faust Application.

Parameters **id** (*str*) – Application ID.

Keyword Arguments **loop** (*asyncio.AbstractEventLoop*) – optional event loop to use.

See also:

Application Parameters – for supported keyword arguments.

SCAN_CATEGORIES = ['faust.agent', 'faust.command', 'faust.page', 'faust.service', 'fau

class BootStrategy(*app: faust.types.app.AppT*, ***, *enable_web: bool = None*, *enable_kafka: bool = None*, *enable_kafka_producer: bool = None*, *enable_kafka_consumer: bool = None*, *enable_sensors: bool = None*) → None

App startup strategy.

The startup strategy defines the graph of services to start when the Faust worker for an app starts.

agents() → Iterable[mode.types.services.ServiceT]

Return list of services required to start agents.

Return type Iterable[ServiceT[]]

client_only() → Iterable[mode.types.services.ServiceT]

Return services to start when app is in client_only mode.

Return type Iterable[ServiceT[]]

enable_kafka = True

enable_kafka_consumer = None

enable_kafka_producer = None

enable_sensors = True

enable_web = None

kafka_client_consumer() → Iterable[mode.types.services.ServiceT]

Return list of services required to start Kafka client consumer.

Return type Iterable[ServiceT[]]

kafka_conductor() → Iterable[mode.types.services.ServiceT]

Return list of services required to start Kafka conductor.

Return type Iterable[ServiceT[]]

kafka_consumer() → Iterable[mode.types.services.ServiceT]

Return list of services required to start Kafka consumer.

Return type Iterable[ServiceT[]]

kafka_producer() → Iterable[mode.types.services.ServiceT]

Return list of services required to start Kafka producer.

Return type Iterable[ServiceT[]]

producer_only() → Iterable[mode.types.services.ServiceT]

Return services to start when app is in producer_only mode.

Return type Iterable[ServiceT[]]

sensors() → Iterable[mode.types.services.ServiceT]

Return list of services required to start sensors.

Return type Iterable[ServiceT[]]

server() → Iterable[mode.types.services.ServiceT]

Return services to start when app is in default mode.

Return type Iterable[ServiceT[]]

tables() → Iterable[mode.types.services.ServiceT]

Return list of table-related services.

Return type Iterable[ServiceT[]]

web_components() → Iterable[mode.types.services.ServiceT]
Return list of web-related services (excluding web server).
Return type Iterable[ServiceT[]]

web_server() → Iterable[mode.types.services.ServiceT]
Return list of web-server services.
Return type Iterable[ServiceT[]]

class Settings(*id: str, *, debug: bool = None, version: int = None, broker: Union[str, yarl.URL, List[yarl.URL]] = None, broker_client_id: str = None, broker_request_timeout: Union[datetime.timedelta, float, str] = None, broker_credentials: Union[faust.types.auth.CredentialsT, ssl.SSLContext] = None, broker_commit_every: int = None, broker_commit_interval: Union[datetime.timedelta, float, str] = None, broker_commit_livelock_soft_timeout: Union[datetime.timedelta, float, str] = None, broker_session_timeout: Union[datetime.timedelta, float, str] = None, broker_heartbeat_interval: Union[datetime.timedelta, float, str] = None, broker_check_crcs: bool = None, broker_max_poll_records: int = None, broker_max_poll_interval: int = None, broker_consumer: Union[str, yarl.URL, List[yarl.URL]] = None, broker_producer: Union[str, yarl.URL, List[yarl.URL]] = None, agent_supervisor: Union[_T, str] = None, store: Union[str, yarl.URL] = None, cache: Union[str, yarl.URL] = None, web: Union[str, yarl.URL] = None, web_enabled: bool = True, processing_guarantee: Union[str, faust.types.enums.ProcessingGuarantee] = None, timezone: datetime.tzinfo = None, autodiscover: Union[bool, Iterable[str], Callable[Iterable[str]]] = None, origin: str = None, canonical_url: Union[str, yarl.URL] = None, datadir: Union[pathlib.Path, str] = None, tabledir: Union[pathlib.Path, str] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, logging_config: Dict = None, loghandlers: List[logging.Handler] = None, table_cleanup_interval: Union[datetime.timedelta, float, str] = None, table_standby_replicas: int = None, table_key_index_size: int = None, topic_replication_factor: int = None, topic_partitions: int = None, topic_allow_declare: bool = None, topic_disable_leader: bool = None, id_format: str = None, reply_to: str = None, reply_to_prefix: str = None, reply_create_topic: bool = None, reply_expires: Union[datetime.timedelta, float, str] = None, ssl_context: ssl.SSLContext = None, stream_buffer_maxsize: int = None, stream_wait_empty: bool = None, stream_ack_cancelled_tasks: bool = None, stream_ack_exceptions: bool = None, stream_publish_on_commit: bool = None, stream_recovery_delay: Union[datetime.timedelta, float, str] = None, producer_linger_ms: int = None, producer_max_batch_size: int = None, producer_acks: int = None, producer_max_request_size: int = None, producer_compression_type: str = None, producer_partitioner: Union[_T, str] = None, producer_request_timeout: Union[datetime.timedelta, float, str] = None, producer_api_version: str = None, consumer_max_fetch_size: int = None, consumer_auto_offset_reset: str = None, web_bind: str = None, web_port: int = None, web_host: str = None, web_transport: Union[str, yarl.URL] = None, web_in_thread: bool = None, web_cors_options: Mapping[str, faust.types.web.ResourceOptions] = None, worker_redirect_stdouts: bool = None, worker_redirect_stdouts_level: Union[int, str] = None, Agent: Union[_T, str] = None, ConsumerScheduler: Union[_T, str] = None, Event: Union[_T, str] = None, Schema: Union[_T, str] = None, Stream: Union[_T, str] = None, Table: Union[_T, str] = None, SetTable: Union[_T, str] = None, GlobalTable: Union[_T, str] = None, SetGlobalTable: Union[_T, str] = None, TableManager: Union[_T, str] = None, Serializers: Union[_T, str] = None, Worker: Union[_T, str] = None, PartitionAssignor: Union[_T, str] = None, LeaderAssignor: Union[_T, str] = None, Router: Union[_T, str] = None, Topic: Union[_T, str] = None, HttpClient: Union[_T, str] = None, Monitor: Union[_T, str] = None, url: Union[str, yarl.URL] = None, **kwargs: Any*) → None

property Agent

Return type Type[*AgentT*[]]

property ConsumerScheduler

Return type Type[SchedulingStrategyT]

property Event
Return type Type[*EventT*[]]

property GlobalTable
Return type Type[*GlobalTableT*[]]

property HttpClient
Return type Type[ClientSession]

property LeaderAssignor
Return type Type[*LeaderAssignorT*[]]

property Monitor
Return type Type[*SensorT*[]]

property PartitionAssignor
Return type Type[*PartitionAssignorT*]

property Router
Return type Type[*RouterT*]

property Schema
Return type Type[*SchemaT*[~KT, ~VT]]

property Serializers
Return type Type[*RegistryT*]

property SetGlobalTable
Return type Type[*GlobalTableT*[]]

property SetTable
Return type Type[*TableT*[~KT, ~VT]]

property Stream
Return type Type[*StreamT*[+T_co]]

property Table
Return type Type[*TableT*[~KT, ~VT]]

property TableManager
Return type Type[*TableManagerT*[]]

property Topic
Return type Type[*TopicT*[]]

property Worker
Return type Type[_WorkerT]

property agent_supervisor
Return type Type[SupervisorStrategyT]

property appdir
Return type Path

autodiscover = False

property broker
Return type List[URL]

broker_check_crcs = True

broker_client_id = 'faust-1.9.0'

broker_commit_every = 10000

property broker_commit_interval
Return type float

property broker_commit_livelock_soft_timeout
Return type float

property broker_consumer
Return type List[URL]

property broker_credentials
Return type Optional[*CredentialsT*]

property broker_heartbeat_interval
Return type float

broker_max_poll_interval = 1000.0

property broker_max_poll_records
Return type Optional[int]

property broker_producer
Return type List[URL]

property broker_request_timeout
Return type float

property broker_session_timeout
Return type float

property cache
Return type URL

property canonical_url
Return type URL

consumer_auto_offset_reset = 'earliest'

consumer_max_fetch_size = 4194304

property datadir
Return type Path

debug = False

find_old_versiondirs() → Iterable[pathlib.Path]
Return type Iterable[Path]

property id
Return type str

id_format = '{id}-v{self.version}'

key_serializer = 'raw'

logging_config = None

property name
Return type str

property origin
Return type Optional[str]

property processing_guarantee
Return type *ProcessingGuarantee*

producer_acks = -1

producer_api_version = 'auto'

producer_compression_type = None

producer_linger_ms = 0

producer_max_batch_size = 16384

producer_max_request_size = 1000000

property producer_partitioner

Return type Optional[Callable[[Optional[bytes], Sequence[int], Sequence[int]], int]]

property producer_request_timeout

Return type float

reply_create_topic = False

property reply_expires

Return type float

reply_to_prefix = 'f-reply-'

classmethod setting_names() → Set[str]

Return type Set[str]

ssl_context = None

property store

Return type URL

stream_ack_cancelled_tasks = True

stream_ack_exceptions = True

stream_buffer_maxsize = 4096

stream_publish_on_commit = False

property stream_recovery_delay

Return type float

stream_wait_empty = True

property table_cleanup_interval

Return type float

table_key_index_size = 1000

table_standby_replicas = 1

property tabledir

Return type Path

timezone = datetime.timezone.utc

topic_allow_declare = True

topic_disable_leader = False

topic_partitions = 8

topic_replication_factor = 1

value_serializer = 'json'

property version

Return type int

property web
Return type URL

web_bind = '0.0.0.0'

web_cors_options = None

web_host = 'build-10233069-project-230058-faust'

web_in_thread = False

web_port = 6066

property web_transport
Return type URL

worker_redirect_stdouts = True

worker_redirect_stdouts_level = 'WARN'

client_only = False
Set this to True if app should only start the services required to operate as an RPC client (producer and simple reply consumer).

producer_only = False
Set this to True if app should run without consumer/tables.

tracer = None
Optional tracing support.

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of additional service dependencies.

The services returned will be started with the app when the app starts.

Return type Iterable[ServiceT[]]

async on_first_start() → None
Call first time app starts in this process.

Return type None

async on_start() → None
Call every time app start/restarts.

Return type None

async on_started() → None
Call when app is fully started.

Return type None

async on_started_init_extra_tasks() → None
Call when started to start additional tasks.

Return type None

async on_started_init_extra_services() → None
Call when initializing extra services at startup.

Return type None

async on_init_extra_service(*service: Union[mode.types.services.ServiceT, Type[mode.types.services.ServiceT]]*) → mode.types.services.ServiceT

Call when adding user services to this app.

Return type ServiceT[]

config_from_object(*obj: Any*, ***, *silent: bool = False*, *force: bool = False*) → None

Read configuration from object.

Object is either an actual object or the name of a module to import.

Examples

```
>>> app.config_from_object('myproj.faustconfig')
```

```
>>> from myproj import faustconfig
>>> app.config_from_object(faustconfig)
```

Parameters

- **silent** (*bool*) – If true then import errors will be ignored.
- **force** (*bool*) – Force reading configuration immediately. By default the configuration will be read only when required.

Return type None

finalize() → None

Finalize app configuration.

Return type None

worker_init() → None

Init worker/CLI commands.

Return type None

worker_init_post_autodiscover() → None

Init worker after autodiscover.

Return type None

discover(**extra_modules: str*, *categories: Iterable[str] = None*, *ignore: Iterable[Any] = [<built-in method search of _sre.SRE_Pattern object>, '.__main__']*) → None

Discover decorators in packages.

Return type None

main() → NoReturn

Execute the **faust** umbrella command using this app.

Return type _NoReturn

topic(**topics: str, pattern: Union[str, Pattern[~AnyStr]] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, replicas: int = None, acks: bool = True, internal: bool = False, config: Mapping[str, Any] = None, maxsize: int = None, allow_empty: bool = False, has_prefix: bool = False, loop: asyncio.events.AbstractEventLoop = None*) → faust.types.topics.TopicT

Create topic description.

Topics are named channels (for example a Kafka topic), that exist on a server. To make an ephemeral local communication channel use: *channel()*.

See also:

faust.topics.Topic

Return type *TopicT*[]

channel(**, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, maxsize: int = None, loop: asyncio.events.AbstractEventLoop = None*) → faust.types.channels.ChannelT

Create new channel.

By default this will create an in-memory channel used for intra-process communication, but in practice channels can be backed by any transport (network or even means of inter-process communication).

See also:

faust.channels.Channel.

Return type *ChannelT*[]

agent(*channel: Union[str, faust.types.channels.ChannelT] = None, *, name: str = None, concurrency: int = 1, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, isolated_partitions: bool = False, use_reply_headers: bool = False, **kwargs: Any*) → Callable[Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], faust.types.agents.AgentT]

Create Agent from async def function.

It can be a regular async function:

```
@app.agent()
async def my_agent(stream):
    async for number in stream:
        print(f'Received: {number!r}')
```

Or it can be an async iterator that yields values. These values can be used as the reply in an RPC-style call, or for sinks: callbacks that forward events to other agents/topics/statsd, and so on:

```
@app.agent(sink=[log_topic])
async def my_agent(requests):
    async for number in requests:
        yield number * 2
```

Return type Callable[[Callable[[*StreamT*[+T_co]], Union[Coroutine[Any, Any, None], Awaitable[None], AsyncIterable[+T_co]]]], *AgentT*[]]

actor(*channel: Union[str, faust.types.channels.ChannelT] = None, *, name: str = None, concurrency: int = 1, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, isolated_partitions: bool = False, use_reply_headers: bool = False, **kwargs: Any*) → Callable[Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], faust.types.agents.AgentT]

Create Agent from async def function.

It can be a regular async function:

```
@app.agent()
async def my_agent(stream):
    async for number in stream:
        print(f'Received: {number!r}')
```

Or it can be an async iterator that yields values. These values can be used as the reply in an RPC-style call, or for sinks: callbacks that forward events to other agents/topics/statsd, and so on:

```
@app.agent(sink=[log_topic])
async def my_agent(requests):
    async for number in requests:
        yield number * 2
```

Return type Callable[[Callable[[*StreamT*[+T_co]], Union[Coroutine[Any, Any, None], Awaitable[None], AsyncIterable[+T_co]]]], *AgentT*[]]

task(*fun: Union[Callable[AppT, Awaitable], Callable[Awaitable]] = None, *, on_leader: bool = False, traced: bool = True*) → Union[Callable[Union[Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]], Union[Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]]], Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]]

Define an async def function to be started with the app.

This is like *timer()* but a one-shot task only executed at worker startup (after recovery and the worker is fully ready for operation).

The function may take zero, or one argument. If the target function takes an argument, the app argument is passed:

```
>>> @app.task
>>> async def on_startup(app):
...     print('STARTING UP: %r' % (app,))
```

Nullary functions are also supported:

```
>>> @app.task
>>> async def on_startup():
...     print('STARTING UP')
```

Return type Union[Callable[[Union[Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]], Union[Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]], Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]

timer(*interval: Union[datetime.timedelta, float, str], on_leader: bool = False, traced: bool = True, name: str = None, max_drift_correction: float = 0.1*) → Callable

Define an async def function to be run at periodic intervals.

Like *task()*, but executes periodically until the worker is shut down.

This decorator takes an async function and adds it to a list of timers started with the app.

Parameters

- **interval** (*Seconds*) – How often the timer executes in seconds.
- **on_leader** (*bool*) – Should the timer only run on the leader?

Example

```
>>> @app.timer(interval=10.0)
>>> async def every_10_seconds():
...     print('TEN SECONDS JUST PASSED')
```

```
>>> app.timer(interval=5.0, on_leader=True)
>>> async def every_5_seconds():
...     print('FIVE SECONDS JUST PASSED. ALSO, I AM THE LEADER!')
```

Return type Callable

crontab(*cron_format: str*, *, *timezone: datetime.tzinfo = None*, *on_leader: bool = False*, *traced: bool = True*) → Callable

Define periodic task using Crontab description.

This is an `async def` function to be run at the fixed times, defined by the Cron format.

Like *timer()*, but executes at fixed times instead of executing at certain intervals.

This decorator takes an async function and adds it to a list of Cronjobs started with the app.

Parameters **cron_format** (*str*) – The Cron spec defining fixed times to run the decorated function.

Keyword Arguments

- **timezone** – The timezone to be taken into account for the Cron jobs. If not set value from *timezone* will be taken.
- **on_leader** – Should the Cron job only run on the leader?

Example

```
>>> @app.crontab(cron_format='30 18 * * *',
                 timezone=pytz.timezone('US/Pacific'))
>>> async def every_6_30_pm_pacific():
...     print('IT IS 6:30pm')
```

```
>>> app.crontab(cron_format='30 18 * * *', on_leader=True)
>>> async def every_6_30_pm():
...     print('6:30pm UTC; ALSO, I AM THE LEADER!')
```

Return type Callable

service (*cls: Type[mode.types.services.ServiceT]*) → Type[mode.types.services.ServiceT]
Decorate `mode.Service` to be started with the app.

Examples

```
from mode import Service

@app.service
class Foo(Service):
    ...
```

Return type `Type`[`ServiceT`[]]

is_leader () → bool
Return `True` if we are in leader worker process.

Return type `bool`

stream (*channel: Union[AsyncIterable, Iterable], beacon: mode.utils.types.trees.NodeT = None, **kwargs: Any*) → faust.types.streams.StreamT
Create new stream from channel/topic/iterable/async iterable.

Parameters

- **channel** (`Union`[`AsyncIterable`[+T_co], `Iterable`[+T_co]]) – Iterable to stream over (async or non-async).
- **kwargs** (`Any`) – See *`Stream`*.

Return type *`StreamT`*[+T_co]

Returns to iterate over events in the stream.

Return type *faust.Stream*

Table (*name: str, *, default: Callable[Any] = None, window: faust.types.windows.WindowT = None, partitions: int = None, help: str = None, **kwargs: Any*) → faust.types.tables.TableT
Define new table.

Parameters

- **name** (`str`) – Name used for table, note that two tables living in the same application cannot have the same name.
- **default** (`Optional`[`Callable`[[], `Any`]]) – A callable, or type that will return a default value for keys missing in this table.
- **window** (`Optional`[*`WindowT`*]) – A windowing strategy to wrap this window in.

Examples

```
>>> table = app.Table('user_to_amount', default=int)
>>> table['George']
0
>>> table['Elaine'] += 1
>>> table['Elaine'] += 1
>>> table['Elaine']
2
```

Return type *TableT*[~KT, ~VT]

GlobalTable(*name: str*, ***, *default: Callable[Any] = None*, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.GlobalTableT

Define new global table.

Parameters

- **name** (*str*) – Name used for global table, note that two global tables living in the same application cannot have the same name.
- **default** (*Optional*[*Callable*[[], *Any*]]) – A callable, or type that will return a default valu for keys missing in this global table.
- **window** (*Optional*[*WindowT*]) – A windowing strategy to wrap this window in.

Examples

```
>>> gtable = app.GlobalTable('user_to_amount', default=int)
>>> gtable['George']
0
>>> gtable['Elaine'] += 1
>>> gtable['Elaine'] += 1
>>> gtable['Elaine']
2
```

Return type *GlobalTableT*[]

SetTable(*name: str*, ***, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *start_manager: bool = False*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.TableT

Table of sets.

Return type *TableT*[~KT, ~VT]

SetGlobalTable(*name: str*, ***, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *start_manager: bool = False*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.TableT

Table of sets (global).

Return type *TableT*[~KT, ~VT]

page(*path: str*, ***, *base: Type[faust.web.views.View] = <class 'faust.web.views.View'>*, *cors_options: Mapping[str, faust.types.web.ResourceOptions] = None*, *name: str = None*) → Callable[Union[Type[faust.types.web.View], Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Type[faust.web.views.View]]

Decorate view to be included in the web server.

Return type *Callable*[[*Union*[*Type*[*View*], *Callable*[[*View*, *Request*], *Union*[*Coroutine*[*Any*, *Any*, *Response*], *Awaitable*[*Response*]]], *Callable*[[*View*, *Request*, *Any*, *Any*], *Union*[*Coroutine*[*Any*, *Any*, *Response*], *Awaitable*[*Response*]]]]], *Type*[*View*]]

table_route(*table: faust.types.tables.CollectionT*, *shard_param: str = None*, ***, *query_param: str = None*, *match_info: str = None*, *exact_key: str = None*) → Callable[Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]]

Decorate view method to route request to table key destination.

Return type Callable[[Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]], Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]]

command(**options: Any*, *base: Optional[Type[faust.app.base._AppCommand]] = None*, ***kwargs: Any*) → Callable[Callable, Type[faust.app.base._AppCommand]]

Decorate `async def` function to be used as CLI command.

Return type Callable[[Callable], Type[_AppCommand]]

create_event(*key: Union[bytes, faust.types.core._ModelT, Any, None]*, *value: Union[bytes, faust.types.core._ModelT, Any]*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]*, *message: faust.types.tuples.Message*) → faust.types.events.EventT

Create new *faust.Event* object.

Return type *EventT*[]

async start_client() → None

Start the app in Client-Only mode necessary for RPC requests.

Notes

Once started as a client the app cannot be restarted as Server.

Return type None

async maybe_start_client() → None

Start the app in Client-Only mode if not started as Server.

Return type None

trace(*name: str*, *trace_enabled: bool = True*, ***extra_context: Any*) → ContextManager

Return new trace context to trace operation using OpenTracing.

Return type ContextManager[+T_co]

traced(*fun: Callable*, *name: str = None*, *sample_rate: float = 1.0*, ***context: Any*) → Callable

Decorate function to be traced using the OpenTracing API.

Return type Callable

async send(*channel: Union[faust.types.channels.ChannelT, str], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*) → Awaitable[faust.types.tuples.RecordMetadata]

Send event to channel/topic.

Parameters

- **channel** (Union[*ChannelT*[], str]) – Channel/topic or the name of a topic to send event to.
- **key** (Union[bytes, _ModelT, Any, None]) – Message key.
- **value** (Union[bytes, _ModelT, Any, None]) – Message value.
- **partition** (Optional[int]) – Specific partition to send to. If not set the partition will be chosen by the partitioner.
- **timestamp** (Optional[float]) – Epoch seconds (from Jan 1 1970 UTC) to use as the message timestamp. Defaults to current time.
- **headers** (Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]) – Mapping of key/value pairs, or iterable of key value pairs to use as headers for the message.
- **schema** (Optional[*SchemaT*[~KT, ~VT]]) – *Schema* to use for serialization.
- **key_serializer** (Union[*CodecT*, str, None]) – Serializer to use (if value is not model). Overrides schema if one is specified.
- **value_serializer** (Union[*CodecT*, str, None]) – Serializer to use (if value is not model). Overrides schema if one is specified.
- **callback** (Optional[Callable[[*FutureMessage*[]], Union[None, Awaitable[None]]]]) – Called after the message is fully delivered to the channel, but not to the consumer. Signature must be unary as the *FutureMessage* future is passed to it.

  The resulting *faust.types.tuples.RecordMetadata* object is then available as `fut.result()`.

Return type Awaitable[*RecordMetadata*]

in_transaction

Return `True` if stream is using transactions.

LiveCheck(***kwargs: Any*) → faust.app.base._LiveCheck

Return new LiveCheck instance testing features for this app.

Return type _LiveCheck

maybe_start_producer

Ensure producer is started. :rtype: *ProducerT*[]

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]]*) → bool

Commit offset for acked messages in specified topics'.

> **Warning:** This will commit acked messages in **all topics** if the topics argument is passed in as `None`.

Return type bool

async on_stop() → None
Call when application stops.

Tip: Remember to call `super` if you override this method.

Return type None

on_rebalance_start() → None
Call when rebalancing starts.

Return type None

on_rebalance_return() → None

Return type None

on_rebalance_end() → None
Call when rebalancing is done.

Return type None

FlowControlQueue(*maxsize: int = None*, ***, *clear_on_resume: bool = False*, *loop: asyncio.events.AbstractEventLoop = None*) → mode.utils.queues.ThrowableQueue
Like `asyncio.Queue`, but can be suspended/resumed.

Return type ThrowableQueue

Worker(***kwargs: Any*) → faust.app.base._Worker
Return application worker instance.

Return type _Worker

on_webserver_init(*web: faust.types.web.Web*) → None
Call when the Web server is initializing.

Return type None

property conf
Application configuration. :rtype: *Settings*

property producer
Message producer. :rtype: *ProducerT*[]

property consumer
Message consumer. :rtype: *ConsumerT*[]

property transport
Consumer message transport. :rtype: *TransportT*

logger = <Logger faust.app.base (WARNING)>

property producer_transport
Producer message transport. :rtype: *TransportT*

property cache
Cache backend. :rtype: *CacheBackendT*[]

tables
Map of available tables, and the table manager service.

topics
Topic Conductor.

This is the mediator that moves messages fetched by the Consumer into the streams.

It's also a set of registered topics by string topic name, so you can check if a topic is being consumed from by doing `topic in app.topics`.

property monitor
Monitor keeps stats about what's going on inside the worker. :rtype: *Monitor*[]

flow_control
Flow control of streams.

This object controls flow into stream queues, and can also clear all buffers.

property http_client
HTTP client Session. :rtype: `ClientSession`

assignor
Partition Assignor.

Responsible for partition assignment.

router
Find the node partitioned data belongs to.

The router helps us route web requests to the wanted Faust node. If a topic is sharded by account_id, the router can send us to the Faust worker responsible for any account. Used by the `@app.table_route` decorator.

web
Web driver.

serializers
Return serializer registry.

property label
Return human readable description of application. :rtype: `str`

property shortlabel
Return short description of application. :rtype: `str`

class faust.**GSSAPICredentials**(*, *kerberos_service_name: str = 'kafka'*, *kerberos_domain_name: str = None*, *ssl_context: ssl.SSLContext = None*, *mechanism: Union[str, faust.types.auth.SASLMechanism] = None*) → None
Describe GSSAPI credentials over SASL.

protocol = 'SASL_PLAINTEXT'

mechanism = 'GSSAPI'

class faust.**SASLCredentials**(*, *username: str = None*, *password: str = None*, *ssl_context: ssl.SSLContext = None*, *mechanism: Union[str, faust.types.auth.SASLMechanism] = None*) → None
Describe SASL credentials.

protocol = 'SASL_PLAINTEXT'

mechanism = 'PLAIN'

class faust.**SSLCredentials**(*context: ssl.SSLContext = None*, *, *purpose: Any = None*, *cafile: Optional[str] = None*, *capath: Optional[str] = None*, *cadata: Optional[str] = None*) → None
Describe SSL credentials/settings.

protocol = 'SSL'

class faust.**Channel**(*app: faust.types.app.AppT, *, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, is_iterator: bool = False, queue: mode.utils.queues.ThrowableQueue = None, maxsize: int = None, root: faust.types.channels.ChannelT = None, active_partitions: Set[faust.types.tuples.TP] = None, loop: asyncio.events.AbstractEventLoop = None*) → None

Create new channel.

Parameters

- **app** (*AppT*[]) – The app that created this channel (`app.channel()`)
- **schema** (Optional[*SchemaT*[~KT, ~VT]]) – Schema used for serialization/deserialization
- **key_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – The Model used for keys in this channel. (overrides schema if one is defined)
- **value_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – The Model used for values in this channel. (overrides schema if one is defined)
- **maxsize** (Optional[int]) – The maximum number of messages this channel can hold. If exceeded any new `put` call will block until a message is removed from the channel.
- **is_iterator** (bool) – When streams iterate over a channel they will call `stream.clone(is_iterator=True)` so this attribute denotes that this channel instance is currently being iterated over.
- **active_partitions** (Optional[Set[*TP*]]) – Set of active topic partitions this channel instance is assigned to.
- **loop** (Optional[AbstractEventLoop]) – The `asyncio` event loop to use.

property queue

Return the underlying queue/buffer backing this channel. :rtype: `ThrowableQueue`

clone(**, is_iterator: bool = None, **kwargs: Any*) → faust.types.channels.ChannelT

Create clone of this channel.

Parameters **is_iterator** (Optional[bool]) – Set to True if this is now a channel that is being iterated over.

Keyword Arguments ****kwargs** – Any keyword arguments passed will override any of the arguments supported by *Channel.__init__*.

Return type *ChannelT*[]

clone_using_queue(*queue: asyncio.queues.Queue*) → faust.types.channels.ChannelT

Create clone of this channel using specific queue instance.

Return type *ChannelT*[]

stream(***kwargs: Any*) → faust.types.streams.StreamT

Create stream reading from this channel.

Return type *StreamT*[+T_co]

get_topic_name() → str

Get the topic name, or raise if this is not a named channel.

Return type str

async send(*, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False) → Awaitable[faust.types.tuples.RecordMetadata]

Send message to channel.

Return type Awaitable[*RecordMetadata*]

send_soon(*, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False, eager_partitioning: bool = False) → faust.types.tuples.FutureMessage

Produce message by adding to buffer.

This method is only supported by *Topic*.

Raises **NotImplementedError** – always for in-memory channel.

Return type *FutureMessage*[]

as_future_message(key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, eager_partitioning: bool = False) → faust.types.tuples.FutureMessage

Create promise that message will be transmitted.

Return type *FutureMessage*[]

prepare_headers(headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]) → Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

Prepare `headers` passed before publishing.

Return type Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

async publish_message(fut: faust.types.tuples.FutureMessage, wait: bool = True) → Awaitable[faust.types.tuples.RecordMetadata]

Publish message to channel.

This is the interface used by `topic.send()`, etc. to actually publish the message on the channel after being buffered up or similar.

It takes a `FutureMessage` object, which contains all the information required to send the message, and acts as a promise that is resolved once the message has been fully transmitted.

Return type Awaitable[*RecordMetadata*]

maybe_declare

Declare/create this channel, but only if it doesn't exist. :rtype: `None`

async declare() → None

Declare/create this channel.

This is used to create this channel on a server, if that is required to operate it.

Return type None

prepare_key(*key: Union[bytes, faust.types.core._ModelT, Any, None], key_serializer: Union[faust.types.codecs.CodecT, str, None], schema: faust.types.serializers.SchemaT = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None] = None*) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Prepare key before it is sent to this channel.

Topic uses this to implement serialization of keys sent to the channel.

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

prepare_value(*value: Union[bytes, faust.types.core._ModelT, Any], value_serializer: Union[faust.types.codecs.CodecT, str, None], schema: faust.types.serializers.SchemaT = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None] = None*) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Prepare value before it is sent to this channel.

Topic uses this to implement serialization of values sent to the channel.

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

async decode(*message: faust.types.tuples.Message, *, propagate: bool = False*) → faust.types.events.EventT

Decode Message into *Event*.

Return type *EventT*[]

async deliver(*message: faust.types.tuples.Message*) → None

Deliver message to queue from consumer.

This is called by the consumer to deliver the message to the channel.

Return type None

async put(*value: Any*) → None

Put event onto this channel.

Return type None

async get(**, timeout: Union[datetime.timedelta, float, str] = None*) → Any

Get the next *Event* received on this channel.

Return type Any

empty() → bool

Return True if the queue is empty.

Return type bool

async on_key_decode_error(*exc: Exception, message: faust.types.tuples.Message*) → None

Unable to decode the key of an item in the queue.

See also:

on_decode_error()

Return type None

async on_value_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None
Unable to decode the value of an item in the queue.

See also:

on_decode_error()

Return type None

async on_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None
Signal that there was an error reading an event in the queue.

When a message in the channel needs deserialization to be reconstructed back to its original form, we will sometimes see decoding/deserialization errors being raised, from missing fields or malformed payloads, and so on.

We will log the exception, but you can also override this to perform additional actions.

Admonition: Kafka In the event a deserialization error occurs, we HAVE to commit the offset of the source message to continue processing the stream.

For this reason it is important that you keep a close eye on error logs. For easy of use, we suggest using log aggregation software, such as Sentry, to surface these errors to your operations team.

Return type None

on_stop_iteration() → None
Signal that iteration over this channel was stopped.

Tip: Remember to call `super` when overriding this method.

Return type None

derive(***kwargs: Any*) → faust.types.channels.ChannelT
Derive new channel from this channel, using new configuration.

See *faust.Topic.derive*.

For local channels this will simply return the same channel.

Return type *ChannelT*[]

async throw(*exc: BaseException*) → None
Throw exception to be received by channel subscribers.

Tip: When you find yourself having to call this from a regular, non-`async def` function, you can use `_throw()` instead.

Return type None

property subscriber_count
Return number of active subscribers to local channel. :rtype: `int`

property label
Short textual description of channel. :rtype: `str`

class faust.**ChannelT**(*app: faust.types.channels._AppT*, ***, *schema: faust.types.channels._SchemaT = None*, *key_type: faust.types.channels._ModelArg = None*, *value_type: faust.types.channels._ModelArg = None*, *is_iterator: bool = False*, *queue: mode.utils.queues.ThrowableQueue = None*, *maxsize: int = None*, *root: Optional[faust.types.channels.ChannelT] = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None

abstract clone(***, *is_iterator: bool = None*, ***kwargs: Any*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract clone_using_queue(*queue: asyncio.queues.Queue*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract stream(***kwargs: Any*) → faust.types.channels._StreamT

Return type _StreamT

abstract get_topic_name() → str

Return type str

abstract async send(***, *key: Union[bytes, faust.types.core._ModelT, Any, None] = None*, *value: Union[bytes, faust.types.core._ModelT, Any] = None*, *partition: int = None*, *timestamp: float = None*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*, *schema: faust.types.channels._SchemaT = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*, *force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract send_soon(***, *key: Union[bytes, faust.types.core._ModelT, Any, None] = None*, *value: Union[bytes, faust.types.core._ModelT, Any] = None*, *partition: int = None*, *timestamp: float = None*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*, *schema: faust.types.channels._SchemaT = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*, *force: bool = False*, *eager_partitioning: bool = False*) → faust.types.tuples.FutureMessage

Return type *FutureMessage*[]

abstract as_future_message(*key: Union[bytes, faust.types.core._ModelT, Any, None] = None*, *value: Union[bytes, faust.types.core._ModelT, Any] = None*, *partition: int = None*, *timestamp: float = None*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*, *schema: faust.types.channels._SchemaT = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*, *eager_partitioning: bool = False*) → faust.types.tuples.FutureMessage

Return type *FutureMessage*[]

abstract async publish_message(*fut: faust.types.tuples.FutureMessage*, *wait: bool = True*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

maybe_declare

Return type None

abstract async declare() → None

Return type None

abstract prepare_key(*key: Union[bytes, faust.types.core._ModelT, Any, None]*, *key_serializer: Union[faust.types.codecs.CodecT, str, None]*, *schema: faust.types.channels._SchemaT = None*) → Any

Return type Any

abstract prepare_value(*value: Union[bytes, faust.types.core._ModelT, Any]*, *value_serializer: Union[faust.types.codecs.CodecT, str, None]*, *schema: faust.types.channels._SchemaT = None*) → Any

Return type Any

abstract async decode(*message: faust.types.tuples.Message*, ***, *propagate: bool = False*) → faust.types.channels._EventT

Return type _EventT

abstract async deliver(*message: faust.types.tuples.Message*) → None

Return type None

abstract async put(*value: Any*) → None

Return type None

abstract async get(***, *timeout: Union[datetime.timedelta, float, str] = None*) → Any

Return type Any

abstract empty() → bool

Return type bool

abstract async on_key_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None

Return type None

abstract async on_value_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None

Return type None

abstract async on_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None

Return type None

abstract on_stop_iteration() → None

Return type None

abstract async throw(*exc: BaseException*) → None

Return type None

abstract derive(***kwargs: Any*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract property subscriber_count

Return type int

abstract property queue

Return type ThrowableQueue

class faust.**Event**(*app: faust.types.app.AppT, key: Union[bytes, faust.types.core._ModelT, Any, None], value: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], message: faust.types.tuples.Message*) → None

An event received on a channel.

Notes

- Events have a key and a value:

```
event.key, event.value
```

- They also have a reference to the original message (if available), such as a Kafka record:

  event.message.offset

- Iterating over channels/topics yields Event:

  async for event in channel: …

- Iterating over a stream (that in turn iterate over channel) yields Event.value:

```
async for value in channel.stream()  # value is event.value
    ...
```

- If you only have a Stream object, you can also access underlying events by using `Stream.events`.

  For example:

```
async for event in channel.stream.events():
    ...
```

  Also commonly used for finding the "current event" related to a value in the stream:

```
stream = channel.stream()
async for event in stream.events():
    event = stream.current_event
    message = event.message
    topic = event.message.topic
```

  You can retrieve the current event in a stream to:

  – Get access to the serialized key+value.

  – Get access to message properties like, what topic+partition the value was received on, or its offset.

  If you want access to both key and value, you should use `stream.items()` instead.

```
async for key, value in stream.items():
    ...
```

`stream.current_event` can also be accessed but you must take extreme care you are using the correct stream object. Methods such as `.group_by(key)` and `.through(topic)` returns cloned stream objects, so in the example:

The best way to access the current_event in an agent is to use the `ContextVar`:

```
from faust import current_event

@app.agent(topic)
async def process(stream):
    async for value in stream:
        event = current_event()
```

app

key

value

message

headers

acked

async send(*channel: Union[str, faust.types.channels.ChannelT], key: Union[bytes, faust.types.core._ModelT, Any, None] = <object object>, value: Union[bytes, faust.types.core._ModelT, Any] = <object object>, partition: int = None, timestamp: float = None, headers: Any = <object object>, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Send object to channel.

Return type `Awaitable[RecordMetadata]`

async forward(*channel: Union[str, faust.types.channels.ChannelT], key: Union[bytes, faust.types.core._ModelT, Any, None] = <object object>, value: Union[bytes, faust.types.core._ModelT, Any] = <object object>, partition: int = None, timestamp: float = None, headers: Any = <object object>, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Forward original message (will not be reserialized).

Return type `Awaitable[RecordMetadata]`

ack() → bool

Acknowledge event as being processed by stream.

When the last stream processor acks the message, the offset in the source topic will be marked as safe-to-commit, and the worker will commit and advance the committed offset.

Return type `bool`

class faust.**EventT**(*app: faust.types.events._AppT, key: Union[bytes, faust.types.core._ModelT, Any, None], value: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], message: faust.types.tuples.Message*) → None

app

key

value

headers

message

acked

abstract async send(*channel: Union[str, faust.types.events._ChannelT], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.events._SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract async forward(*channel: Union[str, faust.types.events._ChannelT], key: Any = None, value: Any = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.events._SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract ack() → bool

Return type bool

class faust.**ModelOptions**(**args*, ***kwargs*)

serializer = None

include_metadata = True

polymorphic_fields = False

allow_blessed_key = False

isodates = False

decimals = False

validation = False

coerce = False

coercions = None

date_parser = None

fields = None

Flattened view of __annotations__ in MRO order.

Type Index

fieldset = None
Set of required field names, for fast argument checking.

Type Index

descriptors = None
Mapping of field name to field descriptor.

Type Index

fieldpos = None
Positional argument index to field name. Used by Record.__init__ to map positional arguments to fields.

Type Index

optionalset = None
Set of optional field names, for fast argument checking.

Type Index

models = None
Mapping of fields that are ModelT

Type Index

modelattrs = None

field_coerce = None
Mapping of fields that need to be coerced. Key is the name of the field, value is the coercion handler function.

Type Index

defaults = None
Mapping of field names to default value.

initfield = None
Mapping of init field conversion callbacks.

polyindex = None
Index of field to polymorphic type

clone_defaults() → faust.types.models.ModelOptions

Return type *ModelOptions*

class faust.**Record** → None
Describes a model type that is a record (Mapping).

Examples

```
>>> class LogEvent(Record, serializer='json'):
...     severity: str
...     message: str
...     timestamp: float
...     optional_field: str = 'default value'
```

```
>>> event = LogEvent(
...     severity='error',
...     message='Broken pact',
...     timestamp=666.0,
... )
```

```
>>> event.severity
'error'
```

```
>>> serialized = event.dumps()
'{"severity": "error", "message": "Broken pact", "timestamp": 666.0}'
```

```
>>> restored = LogEvent.loads(serialized)
<LogEvent: severity='error', message='Broken pact', timestamp=666.0>
```

```
>>> # You can also subclass a Record to create a new record
>>> # with additional fields
>>> class RemoteLogEvent(LogEvent):
...     url: str
```

```
>>> # You can also refer to record fields and pass them around:
>>> LogEvent.severity
>>> <FieldDescriptor: LogEvent.severity (str)>
```

classmethod from_data(*data: Mapping*, ***, *preferred_type: Type[faust.types.models.ModelT] = None*) → faust.models.record.Record

Create model object from Python dictionary.

Return type *Record*

to_representation() → Mapping[str, Any]

Convert model to its Python generic counterpart.

Records will be converted to dictionary.

Return type Mapping[str, Any]

asdict() → Dict[str, Any]

Convert record to Python dictionary.

Return type Dict[str, Any]

class faust.**Monitor**(***, *max_avg_history: int = None*, *max_commit_latency_history: int = None*, *max_send_latency_history: int = None*, *max_assignment_latency_history: int = None*, *messages_sent: int = 0*, *tables: MutableMapping[str, faust.sensors.monitor.TableState] = None*, *messages_active: int = 0*, *events_active: int = 0*, *messages_received_total: int = 0*, *messages_received_by_topic: Counter[str] = None*, *events_total: int = 0*, *events_by_stream: Counter[faust.types.streams.StreamT] = None*, *events_by_task: Counter[_asyncio.Task] = None*, *events_runtime: Deque[float] = None*, *commit_latency: Deque[float] = None*, *send_latency: Deque[float] = None*, *assignment_latency: Deque[float] = None*, *events_s: int = 0*, *messages_s: int = 0*, *events_runtime_avg: float = 0.0*, *topic_buffer_full: Counter[faust.types.topics.TopicT] = None*, *rebalances: int = None*, *rebalance_return_latency: Deque[float] = None*, *rebalance_end_latency: Deque[float] = None*, *rebalance_return_avg: float = 0.0*, *rebalance_end_avg: float = 0.0*, *time: Callable[float] = <built-in function monotonic>*, *http_response_codes: Counter[http.HTTPStatus] = None*, *http_response_latency: Deque[float] = None*, *http_response_latency_avg: float = 0.0*, ***kwargs: Any*) → None

Default Faust Sensor.

This is the default sensor, recording statistics about events, etc.

send_errors = 0

Number of produce operations that ended in error.

assignments_completed = 0
Number of partition assignments completed.

assignments_failed = 0
Number of partitions assignments that failed.

max_avg_history = 100
Max number of total run time values to keep to build average.

max_commit_latency_history = 30
Max number of commit latency numbers to keep.

max_send_latency_history = 30
Max number of send latency numbers to keep.

max_assignment_latency_history = 30
Max number of assignment latency numbers to keep.

rebalances = 0
Number of rebalances seen by this worker.

tables = None
Mapping of tables

commit_latency = None
Deque of commit latency values

send_latency = None
Deque of send latency values

assignment_latency = None
Deque of assignment latency values.

rebalance_return_latency = None
Deque of previous n rebalance return latencies.

rebalance_end_latency = None
Deque of previous n rebalance end latencies.

rebalance_return_avg = 0.0
Average rebalance return latency.

rebalance_end_avg = 0.0
Average rebalance end latency.

messages_active = 0
Number of messages currently being processed.

messages_received_total = 0
Number of messages processed in total.

messages_received_by_topic = None
Count of messages received by topic

messages_sent = 0
Number of messages sent in total.

messages_sent_by_topic = None
Number of messages sent by topic.

messages_s = 0
Number of messages being processed this second.

events_active = 0
Number of events currently being processed.

events_total = 0
Number of events processed in total.

events_by_task = None
Count of events processed by task

events_by_stream = None
Count of events processed by stream

events_s = 0
Number of events being processed this second.

events_runtime_avg = 0.0
Average event runtime over the last second.

events_runtime = None
Deque of run times used for averages

topic_buffer_full = None
Counter of times a topics buffer was full

http_response_codes = None
Counter of returned HTTP status codes.

http_response_latency = None
Deque of previous n HTTP request->response latencies.

http_response_latency_avg = 0.0
Average request->response latency.

metric_counts = None
Arbitrary counts added by apps

tp_committed_offsets = None
Last committed offsets by TopicPartition

tp_read_offsets = None
Last read offsets by TopicPartition

tp_end_offsets = None
Log end offsets by TopicPartition

secs_since(*start_time: float*) → float
Given timestamp start, return number of seconds since that time.

Return type float

ms_since(*start_time: float*) → float
Given timestamp start, return number of ms since that time.

Return type float

logger = <Logger faust.sensors.monitor (WARNING)>

secs_to_ms(*timestamp: float*) → float
Convert seconds to milliseconds.

Return type float

asdict() → Mapping
Return monitor state as dictionary.

Return type Mapping[~KT, +VT_co]

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call before message is delegated to streams.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Call when stream starts processing an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Call when stream is done processing an event.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None
Call when conductor topic buffer is full and has to wait.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call when message is fully acknowledged and can be committed.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when value in table is retrieved.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Call when new value for key in table is set.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when key in a table is deleted.

Return type None

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Consumer is about to commit topic offset.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Call when consumer commit offset operation completed.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
Call when message added to producer buffer.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Call when producer finished sending message.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Call when producer was unable to publish message.

Return type None

count(*metric_name: str*, *count: int = 1*) → None
Count metric by name.

Return type None

on_tp_commit(*tp_offsets: MutableMapping[faust.types.tuples.TP, int]*) → None
Call when offset in topic partition is committed.

Return type None

track_tp_end_offset(*tp: faust.types.tuples.TP*, *offset: int*) → None
Track new topic partition end offset for monitoring lags.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, ***, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT, request: faust.web.base.Request, response: Optional[faust.web.base.Response], state: Dict, *, view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

class faust.**Sensor**(***, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None
Base class for sensors.

This sensor does not do anything at all, but can be subclassed to create new monitors.

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Message received by a consumer.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Message sent to a stream as an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Event was acknowledged by stream.

Notes

Acknowledged means a stream finished processing the event, but given that multiple streams may be handling the same event, the message cannot be committed before all streams have processed it. When all streams have acknowledged the event, it will go through *on_message_out()* just before offsets are committed.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
All streams finished processing message.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None
Topic buffer full so conductor had to wait.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Key retrieved from table.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Value set for key in table.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Key deleted from table.

Return type None

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Consumer is about to commit topic offset.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Consumer finished committing topic offset.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
About to send a message.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Message successfully sent.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Error while sending message.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, ***, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT, request: faust.web.base.Request, response: Optional[faust.web.base.Response], state: Dict, *, view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

asdict() → Mapping
Convert sensor state to dictionary.

Return type Mapping[~KT, +VT_co]

logger = <Logger faust.sensors.base (WARNING)>

class faust.**Codec**(*children: Tuple[faust.types.codecs.CodecT, ...] = None*, ***kwargs: Any*) → None
Base class for codecs.

children = None
next steps in the recursive codec chain. `x = pickle | binary` returns codec with children set to `(pickle, binary)`.

nodes = None
cached version of children including this codec as the first node. could use chain below, but seems premature so just copying the list.

kwargs = None
subclasses can support keyword arguments, the base implementation of *clone()* uses this to preserve keyword arguments in copies.

dumps(*obj: Any*) → bytes
Encode object `obj`.

Return type bytes

loads(*s: bytes*) → Any
Decode object from string.

Return type Any

clone(**children: faust.types.codecs.CodecT*) → faust.types.codecs.CodecT
Create a clone of this codec, with optional children added.

Return type *CodecT*

class faust.**Schema**(**, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, allow_empty: bool = None*) → None

update(**, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, allow_empty: bool = None*) → None

Return type None

loads_key(*app: faust.types.app.AppT*, *message: faust.types.tuples.Message*, ***, *loads: Callable = None*, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → KT

Return type ~KT

loads_value(*app: faust.types.app.AppT*, *message: faust.types.tuples.Message*, ***, *loads: Callable = None*, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → VT

Return type ~VT

dumps_key(*app: faust.types.app.AppT, key: Union[bytes, faust.types.core._ModelT, Any, None], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]*) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

dumps_value(*app: faust.types.app.AppT, value: Union[bytes, faust.types.core._ModelT, Any], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]*) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

on_dumps_key_prepare_headers(*key: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]*) → Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

Return type Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

on_dumps_value_prepare_headers(*value: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]*) → Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

Return type Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

async decode(*app: faust.types.app.AppT*, *message: faust.types.tuples.Message*, ***, *propagate: bool = False*) → faust.types.events.EventT

Decode message from topic (compiled function not cached).

Return type *EventT*[]

compile(*app: faust.types.app.AppT*, ***, *on_key_decode_error: Callable[[Exception, faust.types.tuples.Message], Awaitable[None]] = <function _noop_decode_error>*, *on_value_decode_error: Callable[[Exception, faust.types.tuples.Message], Awaitable[None]] = <function _noop_decode_error>*, *default_propagate: bool = False*) → Callable[..., Awaitable[faust.types.events.EventT]]

Compile function used to decode event.

Return type Callable[..., Awaitable[*EventT*[]]]

class faust.**Stream**(*channel: AsyncIterator[T_co]*, ***, *app: faust.types.app.AppT*, *processors: Iterable[Callable[T]] = None*, *combined: List[faust.types.streams.JoinableT] = None*, *on_start: Callable = None*, *join_strategy: faust.types.joins.JoinT = None*, *beacon: mode.utils.types.trees.NodeT = None*, *concurrency_index: int = None*, *prev: faust.types.streams.StreamT = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, *enable_acks: bool = True*, *prefix: str = ''*, *loop: asyncio.events.AbstractEventLoop = None*) → None

A stream: async iterator processing events in channels/topics.

logger = <Logger faust.streams (WARNING)>

mundane_level = 'debug'

get_active_stream() → faust.types.streams.StreamT

Return the currently active stream.

A stream can be derived using `Stream.group_by` etc, so if this stream was used to create another derived stream, this function will return the stream being actively consumed from. E.g. in the example:

```
>>> @app.agent()
... async def agent(a):
...     a = a
...     b = a.group_by(Withdrawal.account_id)
...     c = b.through('backup_topic')
...     async for value in c:
...         ...
```

The return value of `a.get_active_stream()` would be `c`.

Notes

The chain of streams that leads to the active stream is decided by the `_next` attribute. To get to the active stream we just traverse this linked-list:

```
>>> def get_active_stream(self):
...     node = self
...     while node._next:
...         node = node._next
```

Return type *StreamT*[+T_co]

get_root_stream() → faust.types.streams.StreamT
Get the root stream that this stream was derived from.

Return type *StreamT*[+T_co]

add_processor(*processor: Callable[T]*) → None
Add processor callback executed whenever a new event is received.

Processor functions can be async or non-async, must accept a single argument, and should return the value, mutated or not.

For example a processor handling a stream of numbers may modify the value:

```
def double(value: int) -> int:
    return value * 2

stream.add_processor(double)
```

Return type None

info() → Mapping[str, Any]
Return stream settings as a dictionary.

Return type Mapping[str, Any]

clone(***kwargs: Any*) → faust.types.streams.StreamT
Create a clone of this stream.

Notes

If the cloned stream is supposed to supersede this stream, like in `group_by`/`through`/etc., you should use `_chain()` instead so *stream._next = cloned_stream* is set and *get_active_stream()* returns the cloned stream.

Return type *StreamT*[+T_co]

noack() → faust.types.streams.StreamT
Create new stream where acks are manual.

Return type *StreamT*[+T_co]

items() → AsyncIterator[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], T_co]]
Iterate over the stream as `key, value` pairs.

Examples

```
@app.agent(topic)
async def mytask(stream):
    async for key, value in stream.items():
        print(key, value)
```

Return type AsyncIterator[Tuple[Union[bytes, _ModelT, Any, None], +T_co]]

events() → AsyncIterable[faust.types.events.EventT]
Iterate over the stream as events exclusively.

This means the stream must be iterating over a channel, or at least an iterable of event objects.

Return type AsyncIterable[*EventT*[]]

take(*max_: int, within: Union[datetime.timedelta, float, str]*) → AsyncIterable[Sequence[T_co]]
Buffer n values at a time and yield a list of buffered values.

Parameters **within** (Union[timedelta, float, str]) – Timeout for when we give up waiting for another value, and process the values we have. Warning: If there's no timeout (i.e. *timeout=None*), the agent is likely to stall and block buffered events for an unreasonable length of time(!).

Return type AsyncIterable[Sequence[+T_co]]

enumerate(*start: int = 0*) → AsyncIterable[Tuple[int, T_co]]
Enumerate values received on this stream.

Unlike Python's built-in `enumerate`, this works with async generators.

Return type AsyncIterable[Tuple[int, +T_co]]

through(*channel: Union[str, faust.types.channels.ChannelT]*) → faust.types.streams.StreamT
Forward values to in this stream to channel.

Send messages received on this stream to another channel, and return a new stream that consumes from that channel.

Notes

The messages are forwarded after any processors have been applied.

Example

```
topic = app.topic('foo')

@app.agent(topic)
async def mytask(stream):
    async for value in stream.through(app.topic('bar')):
        # value was first received to topic 'foo'
        # then forwarded and reconsumed from topic 'bar'
        print(value)
```

Return type *StreamT*[+T_co]

echo(**channels: Union[str, faust.types.channels.ChannelT]*) → faust.types.streams.StreamT
Forward values to one or more channels.

Unlike *through()*, we don't consume from these channels.

Return type *StreamT*[+T_co]

group_by(*key: Union[faust.types.models.FieldDescriptorT, Callable[T, Union[bytes, faust.types.core._ModelT, Any, None]]], *, name: str = None, topic: faust.types.topics.TopicT = None, partitions: int = None*) → faust.types.streams.StreamT
Create new stream that repartitions the stream using a new key.

Parameters

- **key** (Union[*FieldDescriptorT*[~T], Callable[[~T], Union[bytes, _ModelT, Any, None]]]) – The key argument decides how the new key is generated, it can be a field descriptor, a callable, or an async callable.

  Note: The `name` argument must be provided if the key argument is a callable.

- **name** (Optional[str]) – Suffix to use for repartitioned topics. This argument is required if *key* is a callable.

Examples

Using a field descriptor to use a field in the event as the new key:

```
s = withdrawals_topic.stream()
# values in this stream are of type Withdrawal
async for event in s.group_by(Withdrawal.account_id):
    ...
```

Using an async callable to extract a new key:

```
s = withdrawals_topic.stream()

async def get_key(withdrawal):
    return await aiohttp.get(
        f'http://e.com/resolve_account/{withdrawal.account_id}')

async for event in s.group_by(get_key):
    ...
```

Using a regular callable to extract a new key:

```
s = withdrawals_topic.stream()

def get_key(withdrawal):
    return withdrawal.account_id.upper()

async for event in s.group_by(get_key):
    ...
```

Return type *StreamT*[+T_co]

filter(*fun: Callable[T]*) → faust.types.streams.StreamT
Filter values from stream using callback.

The callback may be a traditional function, lambda function, or an *async def* function.

This method is useful for filtering events before repartitioning a stream.

Examples

```
>>> async for v in stream.filter(lambda: v > 1000).group_by(...):
...     # do something
```

Return type *StreamT*[+T_co]

derive_topic(*name: str, *, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, prefix: str = '', suffix: str = ''*) → faust.types.topics.TopicT

Create Topic description derived from the K/V type of this stream.

Parameters

- **name** (str) – Topic name.
- **key_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – Specific key type to use for this topic. If not set, the key type of this stream will be used.
- **value_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – Specific value type to use for this topic. If not set, the value type of this stream will be used.

Raises **ValueError** – if the stream channel is not a topic.

Return type *TopicT*[]

async throw(*exc: BaseException*) → None

Send exception to stream iteration.

Return type None

combine(**nodes: faust.types.streams.JoinableT, **kwargs: Any*) → faust.types.streams.StreamT

Combine streams and tables into joined stream.

Return type *StreamT*[+T_co]

contribute_to_stream(*active: faust.types.streams.StreamT*) → None

Add stream as node in joined stream.

Return type None

async remove_from_stream(*stream: faust.types.streams.StreamT*) → None

Remove as node in a joined stream.

Return type None

join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT

Create stream where events are joined.

Return type *StreamT*[+T_co]

left_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT

Create stream where events are joined by LEFT JOIN.

Return type *StreamT*[+T_co]

inner_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT

Create stream where events are joined by INNER JOIN.

Return type *StreamT*[+T_co]

outer_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Create stream where events are joined by OUTER JOIN.

Return type *StreamT*[+T_co]

async on_merge(*value: T = None*) → Optional[T]
Signal called when an event is to be joined.

Return type Optional[~T]

async send(*value: T_contra*) → None
Send value into stream locally (bypasses topic).

Return type None

async on_start() → None
Signal called when the stream starts.

Return type None

async stop() → None
Stop this stream.

Return type None

async on_stop() → None
Signal that the stream is stopping.

Return type None

async ack(*event: faust.types.events.EventT*) → bool
Ack event.

This will decrease the reference count of the event message by one, and when the reference count reaches zero, the worker will commit the offset so that the message will not be seen by a worker again.

Parameters **event** (*EventT*[]) – Event to ack.

Return type bool

property label
Return description of stream, used in graphs and logs. :rtype: str

shortlabel
Return short description of stream.

class faust.**StreamT**(*channel: AsyncIterator[T_co] = None, *, app: faust.types.streams._AppT = None, processors: Iterable[Callable[T]] = None, combined: List[faust.types.streams.JoinableT] = None, on_start: Callable = None, join_strategy: faust.types.streams._JoinT = None, beacon: mode.utils.types.trees.NodeT = None, concurrency_index: int = None, prev: Optional[faust.types.streams.StreamT] = None, active_partitions: Set[faust.types.tuples.TP] = None, enable_acks: bool = True, prefix: str = '', loop: asyncio.events.AbstractEventLoop = None*) → None

outbox = None

join_strategy = None

task_owner = None

current_event = None

active_partitions = None

concurrency_index = None

enable_acks = True

prefix = ''

abstract get_active_stream() → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract add_processor(*processor: Callable[T]*) → None

Return type None

abstract info() → Mapping[str, Any]

Return type Mapping[str, Any]

abstract clone(***kwargs: Any*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract async items() → AsyncIterator[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], T_co]]

abstract async events() → AsyncIterable[faust.types.events.EventT]

abstract async take(*max_: int, within: Union[datetime.timedelta, float, str]*) → AsyncIterable[Sequence[T_co]]

abstract enumerate(*start: int = 0*) → AsyncIterable[Tuple[int, T_co]]

Return type AsyncIterable[Tuple[int, +T_co]]

abstract through(*channel: Union[str, faust.types.channels.ChannelT]*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract echo(**channels: Union[str, faust.types.channels.ChannelT]*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract group_by(*key: Union[faust.types.models.FieldDescriptorT, Callable[T, Union[bytes, faust.types.core._ModelT, Any, None]]], *, name: str = None, topic: faust.types.topics.TopicT = None*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract derive_topic(*name: str, *, schema: faust.types.streams._SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, prefix: str = '', suffix: str = ''*) → faust.types.topics.TopicT

Return type *TopicT*[]

abstract async throw(*exc: BaseException*) → None

Return type None

abstract async send(*value: T_contra*) → None

Return type None

abstract async ack(*event: faust.types.events.EventT*) → bool

Return type bool

faust.**current_event**() → Optional[faust.types.events.EventT]

Return the event currently being processed, or None.

Return type Optional[*EventT*[]]

class faust.**GlobalTable**(*app: faust.types.app.AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, recover_callbacks: Set[Callable[Awaitable[None]]] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

logger = <Logger faust.tables.globaltable (WARNING)>

class faust.**Table**(*app: faust.types.app.AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, recover_callbacks: Set[Callable[Awaitable[None]]] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

Table (non-windowed).

class WindowWrapper(*table: faust.types.tables.TableT, *, relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None] = None, key_index: bool = False, key_index_table: faust.types.tables.TableT = None*) → None

Windowed table wrapper.

A windowed table does not return concrete values when keys are accessed, instead `WindowSet` is returned so that the values can be further reduced to the wanted time period.

ValueType

alias of `WindowSet`

as_ansitable(*title: str = '{table.name}', **kwargs: Any*) → str

Draw table as a terminal ANSI table.

Return type str

clone(*relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT

Clone this table using a new time-relativity configuration.

Return type *WindowWrapperT*[]

property get_relative_timestamp

Return the current handler for extracting event timestamp. :rtype: Optional[Callable[[Optional[*EventT*[]]], Union[float, datetime]]]

get_timestamp(*event: faust.types.events.EventT = None*) → float
Get timestamp from event.
Return type float

items(*event: faust.types.events.EventT = None*) → ItemsView
Return table items view: iterate over (key, value) pairs.
Return type ItemsView[~KT, +VT_co]

key_index = False

key_index_table = None

keys() → KeysView
Return table keys view: iterate over keys found in this table.
Return type KeysView[~KT]

property name
Return the name of this table. :rtype: str

on_del_key(*key: Any*) → None
Call when a key is deleted from this table.
Return type None

on_recover(*fun: Callable[Awaitable[None]]*) → Callable[Awaitable[None]]
Call after table recovery.
Return type Callable[[], Awaitable[None]]

on_set_key(*key: Any*, *value: Any*) → None
Call when the value for a key in this table is set.
Return type None

relative_to(*ts: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT
Configure the time-relativity of this windowed table.
Return type *WindowWrapperT*[]

relative_to_field(*field: faust.types.models.FieldDescriptorT*) → faust.types.tables.WindowWrapperT
Configure table to be time-relative to a field in the stream.

This means the window will use the timestamp from the event currently being processed in the stream.

Further it will not use the timestamp of the Kafka message, but a field in the value of the event.

For example a model field:

```
class Account(faust.Record):
    created: float

table = app.Table('foo').hopping(
    ...,
).relative_to_field(Account.created)
```

Return type *WindowWrapperT*[]

relative_to_now() → faust.types.tables.WindowWrapperT
Configure table to be time-relative to the system clock.
Return type *WindowWrapperT*[]

relative_to_stream() → faust.types.tables.WindowWrapperT
Configure table to be time-relative to the stream.

This means the window will use the timestamp from the event currently being processed in the stream.

Return type *WindowWrapperT*[]

values(*event: faust.types.events.EventT = None*) → ValuesView
Return table values view: iterate over values in this table.

Return type ValuesView[+VT_co]

using_window(*window: faust.types.windows.WindowT*, *, *key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table using a specific window type.

Return type *WindowWrapperT*[]

hopping(*size: Union[datetime.timedelta, float, str], step: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table in a hopping window.

Return type *WindowWrapperT*[]

tumbling(*size: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table in a tumbling window.

Return type *WindowWrapperT*[]

on_key_get(*key: KT*) → None
Call when the value for a key in this table is retrieved.

Return type None

on_key_set(*key: KT*, *value: VT*) → None
Call when the value for a key in this table is set.

Return type None

on_key_del(*key: KT*) → None
Call when a key in this table is removed.

Return type None

as_ansitable(*title: str = '{table.name}'*, ***kwargs: Any*) → str
Draw table as a a terminal ANSI table.

Return type str

logger = <Logger faust.tables.table (WARNING)>

class faust.**SetGlobalTable**(*app: faust.types.app.AppT*, *, *start_manager: bool = False*, *manager_topic_name: str = None*, *manager_topic_suffix: str = None*, ***kwargs: Any*) → None

logger = <Logger faust.tables.sets (WARNING)>

class faust.**SetTable**(*app: faust.types.app.AppT*, *, *start_manager: bool = False*, *manager_topic_name: str = None*, *manager_topic_suffix: str = None*, ***kwargs: Any*) → None
Table that maintains a dictionary of sets.

Manager
alias of SetTableManager

WindowWrapper
alias of SetWindowWrapper

logger = <Logger faust.tables.sets (WARNING)>

manager_topic_suffix = '-setmanager'

async on_start() → None
Call when set table starts.

Return type None

class faust.**Topic**(*app: faust.types.app.AppT, *, topics: Sequence[str] = None, pattern: Union[str, Pattern[~AnyStr]] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, is_iterator: bool = False, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, replicas: int = None, acks: bool = True, internal: bool = False, config: Mapping[str, Any] = None, queue: mode.utils.queues.ThrowableQueue = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, maxsize: int = None, root: faust.types.channels.ChannelT = None, active_partitions: Set[faust.types.tuples.TP] = None, allow_empty: bool = None, has_prefix: bool = False, loop: asyncio.events.AbstractEventLoop = None*) → None
Define new topic description.

Parameters

- **app** (*AppT*[]) – App instance used to create this topic description.
- **topics** (Optional[Sequence[str]]) – List of topic names.
- **partitions** (Optional[int]) – Number of partitions for these topics. On declaration, topics are created using this. Note: If a message is produced before the topic is declared, and autoCreateTopics is enabled on the Kafka Server, the number of partitions used will be specified by the server configuration.
- **retention** (Union[timedelta, float, str, None]) – Number of seconds (as float/timedelta) to keep messages in the topic before they can be expired by the server.
- **pattern** (Union[str, Pattern[AnyStr], None]) – Regular expression evaluated to decide what topics to subscribe to. You cannot specify both topics and a pattern.
- **schema** (Optional[*SchemaT*[~KT, ~VT]]) – Schema used for serialization/deserialization.
- **key_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – How to deserialize keys for messages in this topic. Can be a faust.Model type, str, bytes, or None for "autodetect" (Overrides schema if one is defined).
- **value_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – How to deserialize values for messages in this topic. Can be a faust.Model type, str, bytes, or None for "autodetect" (Overrides schema if ones is defined).
- **active_partitions** (Optional[Set[*TP*]]) – Set of *faust.types.tuples.TP* that this topic should be restricted to.

Raises **TypeError** – if both *topics* and *pattern* is provided.

async send(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Send message to topic.

Return type Awaitable[*RecordMetadata*]

send_soon(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False, eager_partitioning: bool = False*) → faust.types.tuples.FutureMessage
Produce message by adding to buffer.

Notes

This method can be used by non-*async def* functions to produce messages.

Return type *FutureMessage*[]

async put(*event: faust.types.events.EventT*) → None
Put even directly onto the underlying queue of this topic.

This will only affect subscribers to a particular instance, in a particular process.

Return type None

property pattern
Regular expression used by this topic (if any). :rtype: Optional[Pattern[AnyStr]]

property partitions
Return the number of configured partitions for this topic.

Notes

This is only active for internal topics, fully owned and managed by Faust itself.

We never touch the configuration of a topic that exists in Kafka, and Kafka will sometimes automatically create topics when they don't exist. In this case the number of partitions for the automatically created topic will depend on the Kafka server configuration (`num.partitions`).

Always make sure your topics have the correct number of partitions. :rtype: Optional[int]

derive(***kwargs: Any*) → faust.types.channels.ChannelT
Create topic derived from the configuration of this topic.

Configuration will be copied from this topic, but any parameter overridden as a keyword argument.

See also:

derive_topic(): for a list of supported keyword arguments.

Return type *ChannelT*[]

derive_topic(*, *topics: Sequence[str] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, internal: bool = None, config: Mapping[str, Any] = None, prefix: str = '', suffix: str = '', **kwargs: Any*) → faust.types.topics.TopicT

Create new topic with configuration derived from this topic.

Return type *TopicT*[]

get_topic_name() → str

Return the main topic name of this topic description.

As topic descriptions can have multiple topic names, this will only return when the topic has a singular topic name in the description.

Raises

- **TypeError** – if configured with a regular expression pattern.
- **ValueError** – if configured with multiple topic names.
- **TypeError** – if not configured with any names or patterns.

Return type str

async publish_message(*fut: faust.types.tuples.FutureMessage, wait: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Fulfill promise to publish message to topic.

Return type Awaitable[*RecordMetadata*]

maybe_declare

Declare/create this topic, only if it does not exist. :rtype: None

async declare() → None

Declare/create this topic on the server.

Return type None

class faust.**TopicT**(*app: faust.types.topics._AppT, *, topics: Sequence[str] = None, pattern: Union[str, Pattern[~AnyStr]] = None, schema: faust.types.topics._SchemaT = None, key_type: faust.types.topics._ModelArg = None, value_type: faust.types.topics._ModelArg = None, is_iterator: bool = False, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, replicas: int = None, acks: bool = True, internal: bool = False, config: Mapping[str, Any] = None, queue: mode.utils.queues.ThrowableQueue = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, maxsize: int = None, root: faust.types.channels.ChannelT = None, active_partitions: Set[faust.types.tuples.TP] = None, allow_empty: bool = False, has_prefix: bool = False, loop: asyncio.events.AbstractEventLoop = None*) → None

topics = None

Iterable/Sequence of topic names to subscribe to.

retention = None

expiry time in seconds for messages in the topic.

Type Topic retention setting

compacting = None
Flag that when enabled means the topic can be "compacted": if the topic is a log of key/value pairs, the broker can delete old values for the same key.

replicas = None
Number of replicas for topic.

config = None
Additional configuration as a mapping.

acks = None
Enable acks for this topic.

internal = None
it's owned by us and we are allowed to create or delete the topic as necessary.

Type Mark topic as internal

has_prefix = False

abstract property pattern

Return type Optional[Pattern[AnyStr]]

abstract property partitions

Return type Optional[int]

abstract derive(***kwargs: Any*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract derive_topic(**, topics: Sequence[str] = None, schema: faust.types.topics._SchemaT = None, key_type: faust.types.topics._ModelArg = None, value_type: faust.types.topics._ModelArg = None, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, internal: bool = False, config: Mapping[str, Any] = None, prefix: str = '', suffix: str = '', **kwargs: Any*) → faust.types.topics.TopicT

Return type *TopicT*[]

class faust.**Settings**(*id: str, *, debug: bool = None, version: int = None, broker: Union[str, yarl.URL, List[yarl.URL]] = None, broker_client_id: str = None, broker_request_timeout: Union[datetime.timedelta, float, str] = None, broker_credentials: Union[faust.types.auth.CredentialsT, ssl.SSLContext] = None, broker_commit_every: int = None, broker_commit_interval: Union[datetime.timedelta, float, str] = None, broker_commit_livelock_soft_timeout: Union[datetime.timedelta, float, str] = None, broker_session_timeout: Union[datetime.timedelta, float, str] = None, broker_heartbeat_interval: Union[datetime.timedelta, float, str] = None, broker_check_crcs: bool = None, broker_max_poll_records: int = None, broker_max_poll_interval: int = None, broker_consumer: Union[str, yarl.URL, List[yarl.URL]] = None, broker_producer: Union[str, yarl.URL, List[yarl.URL]] = None, agent_supervisor: Union[_T, str] = None, store: Union[str, yarl.URL] = None, cache: Union[str, yarl.URL] = None, web: Union[str, yarl.URL] = None, web_enabled: bool = True, processing_guarantee: Union[str, faust.types.enums.ProcessingGuarantee] = None, timezone: datetime.tzinfo = None, autodiscover: Union[bool, Iterable[str], Callable[Iterable[str]]] = None, origin: str = None, canonical_url: Union[str, yarl.URL] = None, datadir: Union[pathlib.Path, str] = None, tabledir: Union[pathlib.Path, str] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, logging_config: Dict = None, loghandlers: List[logging.Handler] = None, table_cleanup_interval: Union[datetime.timedelta, float, str] = None, table_standby_replicas: int = None, table_key_index_size: int = None, topic_replication_factor: int = None, topic_partitions: int = None, topic_allow_declare: bool = None, topic_disable_leader: bool = None, id_format: str = None, reply_to: str = None, reply_to_prefix: str = None, reply_create_topic: bool = None, reply_expires: Union[datetime.timedelta, float, str] = None, ssl_context: ssl.SSLContext = None, stream_buffer_maxsize: int = None, stream_wait_empty: bool = None, stream_ack_cancelled_tasks: bool = None, stream_ack_exceptions: bool = None, stream_publish_on_commit: bool = None, stream_recovery_delay: Union[datetime.timedelta, float, str] = None, producer_linger_ms: int = None, producer_max_batch_size: int = None, producer_acks: int = None, producer_max_request_size: int = None, producer_compression_type: str = None, producer_partitioner: Union[_T, str] = None, producer_request_timeout: Union[datetime.timedelta, float, str] = None, producer_api_version: str = None, consumer_max_fetch_size: int = None, consumer_auto_offset_reset: str = None, web_bind: str = None, web_port: int = None, web_host: str = None, web_transport: Union[str, yarl.URL] = None, web_in_thread: bool = None, web_cors_options: Mapping[str, faust.types.web.ResourceOptions] = None, worker_redirect_stdouts: bool = None, worker_redirect_stdouts_level: Union[int, str] = None, Agent: Union[_T, str] = None, ConsumerScheduler: Union[_T, str] = None, Event: Union[_T, str] = None, Schema: Union[_T, str] = None, Stream: Union[_T, str] = None, Table: Union[_T, str] = None, SetTable: Union[_T, str] = None, GlobalTable: Union[_T, str] = None, SetGlobalTable: Union[_T, str] = None, TableManager: Union[_T, str] = None, Serializers: Union[_T, str] = None, Worker: Union[_T, str] = None, PartitionAssignor: Union[_T, str] = None, LeaderAssignor: Union[_T, str] = None, Router: Union[_T, str] = None, Topic: Union[_T, str] = None, HttpClient: Union[_T, str] = None, Monitor: Union[_T, str] = None, url: Union[str, yarl.URL] = None, **kwargs: Any*) → None

classmethod setting_names() → Set[str]

Return type Set[str]

```
id_format = '{id}-v{self.version}'
debug = False
ssl_context = None
autodiscover = False
broker_client_id = 'faust-1.9.0'
timezone = datetime.timezone.utc
broker_commit_every = 10000
broker_check_crcs = True
broker_max_poll_interval = 1000.0
key_serializer = 'raw'
value_serializer = 'json'
table_standby_replicas = 1
table_key_index_size = 1000
topic_replication_factor = 1
topic_partitions = 8
topic_allow_declare = True
topic_disable_leader = False
reply_create_topic = False
logging_config = None
stream_buffer_maxsize = 4096
stream_wait_empty = True
stream_ack_cancelled_tasks = True
stream_ack_exceptions = True
stream_publish_on_commit = False
producer_linger_ms = 0
producer_max_batch_size = 16384
producer_acks = -1
producer_max_request_size = 1000000
producer_compression_type = None
producer_api_version = 'auto'
consumer_max_fetch_size = 4194304
consumer_auto_offset_reset = 'earliest'
web_bind = '0.0.0.0'
web_port = 6066
web_host = 'build-10233069-project-230058-faust'
web_in_thread = False
```

web_cors_options = None

worker_redirect_stdouts = True

worker_redirect_stdouts_level = 'WARN'

reply_to_prefix = 'f-reply-'

property name

Return type str

property id

Return type str

property origin

Return type Optional[str]

property version

Return type int

property broker

Return type List[URL]

property broker_consumer

Return type List[URL]

property broker_producer

Return type List[URL]

property store

Return type URL

property web

Return type URL

property cache

Return type URL

property canonical_url

Return type URL

property datadir

Return type Path

property appdir

Return type Path

find_old_versiondirs() → Iterable[pathlib.Path]

Return type Iterable[Path]

property tabledir

Return type Path

property processing_guarantee

Return type *ProcessingGuarantee*

property broker_credentials

Return type Optional[*CredentialsT*]

property broker_request_timeout

Return type float

property broker_session_timeout

Return type float

property broker_heartbeat_interval

Return type float

property broker_commit_interval

Return type float

property broker_commit_livelock_soft_timeout

Return type float

property broker_max_poll_records

Return type Optional[int]

property producer_partitioner

Return type Optional[Callable[[Optional[bytes], Sequence[int], Sequence[int]], int]]

property producer_request_timeout

Return type float

property table_cleanup_interval

Return type float

property reply_expires

Return type float

property stream_recovery_delay

Return type float

property agent_supervisor

Return type Type[SupervisorStrategyT]

property web_transport

Return type URL

property Agent

Return type Type[*AgentT*[]]

property ConsumerScheduler

Return type Type[SchedulingStrategyT]

property Event

Return type Type[*EventT*[]]

property Schema

Return type Type[*SchemaT*[~KT, ~VT]]

property Stream

Return type Type[*StreamT*[+T_co]]

property Table

Return type Type[*TableT*[~KT, ~VT]]

property SetTable

Return type Type[*TableT*[~KT, ~VT]]

property GlobalTable

Return type Type[*GlobalTableT*[]]

property SetGlobalTable

Return type Type[*GlobalTableT*[]]

property TableManager

Return type Type[*TableManagerT*[]]

property Serializers

Return type Type[*RegistryT*]

property Worker

Return type Type[_WorkerT]

property PartitionAssignor

Return type Type[*PartitionAssignorT*]

property LeaderAssignor

Return type Type[*LeaderAssignorT*[]]

property Router

Return type Type[*RouterT*]

property Topic

Return type Type[*TopicT*[]]

property HttpClient

Return type Type[ClientSession]

property Monitor

Return type Type[*SensorT*[]]

faust.**HoppingWindow**
alias of faust.windows._PyHoppingWindow

class faust.**TumblingWindow**(*size: Union[datetime.timedelta, float, str]*, *expires: Union[datetime.timedelta, float, str] = None*) → None
Tumbling window type.

Fixed-size, non-overlapping, gap-less windows.

faust.**SlidingWindow**
alias of faust.windows._PySlidingWindow

class faust.**Window**(**args*, ***kwargs*)
Base class for window types.

class faust.**Worker**(*app: faust.types.app.AppT*, **services: mode.types.services.ServiceT*, *sensors: Iterable[faust.types.sensors.SensorT] = None*, *debug: bool = False*, *quiet: bool = False*, *loglevel: Union[str, int] = None*, *logfile: Union[str, IO] = None*, *stdout: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *stderr: IO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *blocking_timeout: float = 10.0*, *workdir: Union[pathlib.Path, str] = None*, *console_port: int = 50101*, *loop: asyncio.events.AbstractEventLoop = None*, *redirect_stdouts: bool = None*, *redirect_stdouts_level: int = None*, *logging_config: Dict = None*, ***kwargs: Any*) → None
Worker.

See also:

This is a subclass of `mode.Worker`.

Usage: You can start a worker using:

1) the **`faust worker`** program.

2) instantiating Worker programmatically and calling *execute_from_commandline()*:

```
>>> worker = Worker(app)
>>> worker.execute_from_commandline()
```

3) or if you already have an event loop, calling `await start`, but in that case *you are responsible for gracefully shutting down the event loop*:

```
async def start_worker(worker: Worker) -> None:
    await worker.start()

def manage_loop():
    loop = asyncio.get_event_loop()
    worker = Worker(app, loop=loop)
    try:
        loop.run_until_complete(start_worker(worker)
    finally:
        worker.stop_and_shutdown_loop()
```

Parameters

- **`app`** (*`AppT`*[]) – The Faust app to start.
- **`*services`** – Services to start with worker. This includes application instances to start.
- **`sensors`** (*`Iterable[SensorT]`*) – List of sensors to include.
- **`debug`** (*`bool`*) – Enables debugging mode [disabled by default].
- **`quiet`** (*`bool`*) – Do not output anything to console [disabled by default].
- **`loglevel`** (*`Union[str, int]`*) – Level to use for logging, can be string (one of: CRIT|ERROR|WARN|INFO|DEBUG), or integer.
- **`logfile`** (*`Union[str, IO]`*) – Name of file or a stream to log to.
- **`stdout`** (*`IO`*) – Standard out stream.
- **`stderr`** (*`IO`*) – Standard err stream.

- **blocking_timeout** (*float*) – When `debug` is enabled this sets the timeout for detecting that the event loop is blocked.
- **workdir** (*Union[str, Path]*) – Custom working directory for the process that the worker will change into when started. This working directory change is permanent for the process, or until something else changes the working directory again.
- **loop** (*asyncio.AbstractEventLoop*) – Custom event loop object.

logger = <Logger faust.worker (WARNING)>

app = None
The Faust app started by this worker.

sensors = None
Additional sensors to add to the Faust app.

workdir = None
Current working directory. Note that if passed as an argument to Worker, the worker will change to this directory when started.

spinner = None
Class that displays a terminal progress spinner (see progress).

async on_start() → None
Signal called every time the worker starts.

Return type `None`

async on_startup_finished() → None
Signal called when worker has started.

Return type `None`

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return service dependencies that must start with the worker.

Return type `Iterable[ServiceT[]]`

async on_first_start() → None
Signal called the first time the worker starts.

First time, means this callback is not called if the worker is restarted by an exception being raised.

Return type `None`

change_workdir(*path: pathlib.Path*) → None
Change the current working directory (CWD).

Return type `None`

autodiscover() → None
Autodiscover modules and files to find @agent decorators, etc.

Return type `None`

async on_execute() → None
Signal called when the worker is about to start.

Return type `None`

on_worker_shutdown() → None
Signal called before the worker is shutting down.

Return type `None`

on_setup_root_logger(*logger: logging.Logger*, *level: int*) → None
Signal called when the root logger is being configured.

Return type None

faust.**uuid**() → str
Generate random UUID string.

Shortcut to `str(uuid4())`.

Return type str

faust.auth

Authentication Credentials.

class faust.auth.**Credentials**(**args*, ***kwargs*)
Base class for authentication credentials.

class faust.auth.**SASLCredentials**(*, *username: str = None*, *password: str = None*, *ssl_context: ssl.SSLContext = None*, *mechanism: Union[str, faust.types.auth.SASLMechanism] = None*) → None
Describe SASL credentials.

protocol = 'SASL_PLAINTEXT'

mechanism = 'PLAIN'

class faust.auth.**GSSAPICredentials**(*, *kerberos_service_name: str = 'kafka'*, *kerberos_domain_name: str = None*, *ssl_context: ssl.SSLContext = None*, *mechanism: Union[str, faust.types.auth.SASLMechanism] = None*) → None
Describe GSSAPI credentials over SASL.

protocol = 'SASL_PLAINTEXT'

mechanism = 'GSSAPI'

class faust.auth.**SSLCredentials**(*context: ssl.SSLContext = None*, *, *purpose: Any = None*, *cafile: Optional[str] = None*, *capath: Optional[str] = None*, *cadata: Optional[str] = None*) → None
Describe SSL credentials/settings.

protocol = 'SSL'

faust.exceptions

Faust exceptions.

exception faust.exceptions.**FaustError**
Base-class for all Faust exceptions.

exception faust.exceptions.**FaustWarning**
Base-class for all Faust warnings.

exception faust.exceptions.**NotReady**
Service not started.

exception faust.exceptions.**AlreadyConfiguredWarning**
Trying to configure app after configuration accessed.

exception faust.exceptions.**ImproperlyConfigured**
The library is not configured/installed correctly.

exception faust.exceptions.**DecodeError**
Error while decoding/deserializing message key/value.

exception faust.exceptions.**KeyDecodeError**
Error while decoding/deserializing message key.

exception faust.exceptions.**ValueDecodeError**
Error while decoding/deserializing message value.

exception faust.exceptions.**SameNode**
Exception raised by router when data is located on same node.

exception faust.exceptions.**ProducerSendError**
Error while sending attached messages prior to commit.

exception faust.exceptions.**ConsumerNotStarted**
Error trying to perform operation on consumer not started.

exception faust.exceptions.**PartitionsMismatch**
Number of partitions between related topics differ.

faust.channels

Channel.

A channel is used to send values to streams.

The stream will iterate over incoming events in the channel.

class faust.channels.**Channel**(*app: faust.types.app.AppT, *, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, is_iterator: bool = False, queue: mode.utils.queues.ThrowableQueue = None, maxsize: int = None, root: faust.types.channels.ChannelT = None, active_partitions: Set[faust.types.tuples.TP] = None, loop: asyncio.events.AbstractEventLoop = None*) → None

Create new channel.

Parameters

- **app** (*AppT*[]) – The app that created this channel (`app.channel()`)
- **schema** (Optional[*SchemaT*[~KT, ~VT]]) – Schema used for serialization/deserialization
- **key_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – The Model used for keys in this channel. (overrides schema if one is defined)
- **value_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – The Model used for values in this channel. (overrides schema if one is defined)
- **maxsize** (Optional[int]) – The maximum number of messages this channel can hold. If exceeded any new `put` call will block until a message is removed from the channel.
- **is_iterator** (bool) – When streams iterate over a channel they will call `stream.clone(is_iterator=True)` so this attribute denotes that this channel instance is currently being iterated over.

- **active_partitions** (Optional[Set[*TP*]]) – Set of active topic partitions this channel instance is assigned to.
- **loop** (Optional[AbstractEventLoop]) – The asyncio event loop to use.

property queue
Return the underlying queue/buffer backing this channel. :rtype: ThrowableQueue

clone(*, *is_iterator: bool = None*, ***kwargs: Any*) → faust.types.channels.ChannelT
Create clone of this channel.

Parameters **is_iterator** (Optional[bool]) – Set to True if this is now a channel that is being iterated over.

Keyword Arguments ****kwargs** – Any keyword arguments passed will override any of the arguments supported by *Channel.__init__*.

Return type *ChannelT*[]

clone_using_queue(*queue: asyncio.queues.Queue*) → faust.types.channels.ChannelT
Create clone of this channel using specific queue instance.

Return type *ChannelT*[]

stream(***kwargs: Any*) → faust.types.streams.StreamT
Create stream reading from this channel.

Return type *StreamT*[+T_co]

get_topic_name() → str
Get the topic name, or raise if this is not a named channel.

Return type str

async send(*, *key: Union[bytes, faust.types.core._ModelT, Any, None] = None*, *value: Union[bytes, faust.types.core._ModelT, Any] = None*, *partition: int = None*, *timestamp: float = None*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*, *schema: faust.types.serializers.SchemaT = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*, *force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]
Send message to channel.

Return type Awaitable[*RecordMetadata*]

send_soon(*, *key: Union[bytes, faust.types.core._ModelT, Any, None] = None*, *value: Union[bytes, faust.types.core._ModelT, Any] = None*, *partition: int = None*, *timestamp: float = None*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*, *schema: faust.types.serializers.SchemaT = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*, *force: bool = False*, *eager_partitioning: bool = False*) → faust.types.tuples.FutureMessage
Produce message by adding to buffer.

This method is only supported by *Topic*.

Raises **NotImplementedError** – always for in-memory channel.

Return type *FutureMessage*[]

as_future_message(*key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, eager_partitioning: bool = False*) → faust.types.tuples.FutureMessage

Create promise that message will be transmitted.

Return type *FutureMessage*[]

prepare_headers(*headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]*) → Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

Prepare `headers` passed before publishing.

Return type Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

async publish_message(*fut: faust.types.tuples.FutureMessage, wait: bool = True*) → Awaitable[faust.types.tuples.RecordMetadata]

Publish message to channel.

This is the interface used by `topic.send()`, etc. to actually publish the message on the channel after being buffered up or similar.

It takes a `FutureMessage` object, which contains all the information required to send the message, and acts as a promise that is resolved once the message has been fully transmitted.

Return type Awaitable[*RecordMetadata*]

maybe_declare

Declare/create this channel, but only if it doesn't exist. :rtype: `None`

async declare() → None

Declare/create this channel.

This is used to create this channel on a server, if that is required to operate it.

Return type `None`

prepare_key(*key: Union[bytes, faust.types.core._ModelT, Any, None], key_serializer: Union[faust.types.codecs.CodecT, str, None], schema: faust.types.serializers.SchemaT = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None] = None*) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Prepare key before it is sent to this channel.

Topic uses this to implement serialization of keys sent to the channel.

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

prepare_value(*value: Union[bytes, faust.types.core._ModelT, Any], value_serializer: Union[faust.types.codecs.CodecT, str, None], schema: faust.types.serializers.SchemaT = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None] = None*) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Prepare value before it is sent to this channel.

Topic uses this to implement serialization of values sent to the channel.

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

async decode(*message: faust.types.tuples.Message*, *, *propagate: bool = False*) → faust.types.events.EventT
Decode `Message` into *Event*.

Return type *EventT*[]

async deliver(*message: faust.types.tuples.Message*) → None
Deliver message to queue from consumer.

This is called by the consumer to deliver the message to the channel.

Return type `None`

async put(*value: Any*) → None
Put event onto this channel.

Return type `None`

async get(*, *timeout: Union[datetime.timedelta, float, str] = None*) → Any
Get the next *Event* received on this channel.

Return type `Any`

empty() → bool
Return `True` if the queue is empty.

Return type `bool`

async on_key_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None
Unable to decode the key of an item in the queue.

See also:

on_decode_error()

Return type `None`

async on_value_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None
Unable to decode the value of an item in the queue.

See also:

on_decode_error()

Return type `None`

async on_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None
Signal that there was an error reading an event in the queue.

When a message in the channel needs deserialization to be reconstructed back to its original form, we will sometimes see decoding/deserialization errors being raised, from missing fields or malformed payloads, and so on.

We will log the exception, but you can also override this to perform additional actions.

Admonition: Kafka In the event a deserialization error occurs, we HAVE to commit the offset of the source message to continue processing the stream.

For this reason it is important that you keep a close eye on error logs. For easy of use, we suggest using log aggregation software, such as Sentry, to surface these errors to your operations team.

Return type `None`

on_stop_iteration() → None
Signal that iteration over this channel was stopped.

Tip: Remember to call `super` when overriding this method.

Return type None

derive(**kwargs: Any*) → faust.types.channels.ChannelT
Derive new channel from this channel, using new configuration.

See *faust.Topic.derive*.

For local channels this will simply return the same channel.

Return type *ChannelT*[]

async throw(*exc: BaseException*) → None
Throw exception to be received by channel subscribers.

Tip: When you find yourself having to call this from a regular, non-`async def` function, you can use `_throw()` instead.

Return type None

property subscriber_count
Return number of active subscribers to local channel. :rtype: `int`

property label
Short textual description of channel. :rtype: `str`

faust.events

Events received in streams.

class faust.events.**Event**(*app: faust.types.app.AppT, key: Union[bytes, faust.types.core._ModelT, Any, None], value: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], message: faust.types.tuples.Message*) → None
An event received on a channel.

Notes

- Events have a key and a value:

```
event.key, event.value
```

- They also have a reference to the original message (if available), such as a Kafka record:

  event.message.offset

- Iterating over channels/topics yields Event:

  async for event in channel: …

- Iterating over a stream (that in turn iterate over channel) yields Event.value:

```
async for value in channel.stream()  # value is event.value
    ...
```

- If you only have a Stream object, you can also access underlying events by using `Stream.events`.

  For example:

  ```
  async for event in channel.stream.events():
      ...
  ```

  Also commonly used for finding the "current event" related to a value in the stream:

  ```
  stream = channel.stream()
  async for event in stream.events():
      event = stream.current_event
      message = event.message
      topic = event.message.topic
  ```

  You can retrieve the current event in a stream to:

  - Get access to the serialized key+value.
  - Get access to message properties like, what topic+partition the value was received on, or its offset.

  If you want access to both key and value, you should use `stream.items()` instead.

  ```
  async for key, value in stream.items():
      ...
  ```

  `stream.current_event` can also be accessed but you must take extreme care you are using the correct stream object. Methods such as `.group_by(key)` and `.through(topic)` returns cloned stream objects, so in the example:

  The best way to access the current_event in an agent is to use the `ContextVar`:

  ```
  from faust import current_event

  @app.agent(topic)
  async def process(stream):
      async for value in stream:
          event = current_event()
  ```

app

key

value

message

headers

acked

async send(*channel: Union[str, faust.types.channels.ChannelT], key: Union[bytes, faust.types.core._ModelT, Any, None] = <object object>, value: Union[bytes, faust.types.core._ModelT, Any] = <object object>, partition: int = None, timestamp: float = None, headers: Any = <object object>, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Send object to channel.

Return type Awaitable[*RecordMetadata*]

async forward(*channel: Union[str, faust.types.channels.ChannelT], key: Union[bytes, faust.types.core._ModelT, Any, None] = <object object>, value: Union[bytes, faust.types.core._ModelT, Any] = <object object>, partition: int = None, timestamp: float = None, headers: Any = <object object>, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Forward original message (will not be reserialized).

Return type Awaitable[*RecordMetadata*]

ack() → bool

Acknowledge event as being processed by stream.

When the last stream processor acks the message, the offset in the source topic will be marked as safe-to-commit, and the worker will commit and advance the committed offset.

Return type bool

faust.joins

Join strategies.

class faust.joins.**Join**(**, stream: faust.types.streams.JoinableT, fields: Tuple[faust.types.models.FieldDescriptorT, ...]*) → None

Base class for join strategies.

async process(*event: faust.types.events.EventT*) → Optional[faust.types.events.EventT]

Process event to be joined with another event.

Return type Optional[*EventT*[]]

class faust.joins.**RightJoin**(**, stream: faust.types.streams.JoinableT, fields: Tuple[faust.types.models.FieldDescriptorT, ...]*) → None

Right-join strategy.

class faust.joins.**LeftJoin**(**, stream: faust.types.streams.JoinableT, fields: Tuple[faust.types.models.FieldDescriptorT, ...]*) → None

Left-join strategy.

class faust.joins.**InnerJoin**(**, stream: faust.types.streams.JoinableT, fields: Tuple[faust.types.models.FieldDescriptorT, ...]*) → None

Inner-join strategy.

class faust.joins.**OuterJoin**(**, stream: faust.types.streams.JoinableT, fields: Tuple[faust.types.models.FieldDescriptorT, ...]*) → None

Outer-join strategy.

faust.streams

Streams.

faust.streams.**current_event**() → Optional[faust.types.events.EventT]
Return the event currently being processed, or None.

Return type Optional[*EventT*[]]

class faust.streams.**Stream**(*channel: AsyncIterator[T_co], *, app: faust.types.app.AppT, processors: Iterable[Callable[T]] = None, combined: List[faust.types.streams.JoinableT] = None, on_start: Callable = None, join_strategy: faust.types.joins.JoinT = None, beacon: mode.utils.types.trees.NodeT = None, concurrency_index: int = None, prev: faust.types.streams.StreamT = None, active_partitions: Set[faust.types.tuples.TP] = None, enable_acks: bool = True, prefix: str = '', loop: asyncio.events.AbstractEventLoop = None*) → None
A stream: async iterator processing events in channels/topics.

logger = <Logger faust.streams (WARNING)>

mundane_level = 'debug'

get_active_stream() → faust.types.streams.StreamT
Return the currently active stream.

A stream can be derived using `Stream.group_by` etc, so if this stream was used to create another derived stream, this function will return the stream being actively consumed from. E.g. in the example:

```
>>> @app.agent()
... async def agent(a):
..      a = a
...     b = a.group_by(Withdrawal.account_id)
...     c = b.through('backup_topic')
...     async for value in c:
...         ...
```

The return value of `a.get_active_stream()` would be `c`.

Notes

The chain of streams that leads to the active stream is decided by the `_next` attribute. To get to the active stream we just traverse this linked-list:

```
>>> def get_active_stream(self):
...     node = self
...     while node._next:
...         node = node._next
```

Return type *StreamT*[+T_co]

get_root_stream() → faust.types.streams.StreamT
Get the root stream that this stream was derived from.

Return type *StreamT*[+T_co]

add_processor(*processor: Callable[T]*) → None
Add processor callback executed whenever a new event is received.

Processor functions can be async or non-async, must accept a single argument, and should return the value, mutated or not.

For example a processor handling a stream of numbers may modify the value:

```
def double(value: int) -> int:
    return value * 2

stream.add_processor(double)
```

Return type None

info() → Mapping[str, Any]
Return stream settings as a dictionary.

Return type Mapping[str, Any]

clone(***kwargs: Any*) → faust.types.streams.StreamT
Create a clone of this stream.

Notes

If the cloned stream is supposed to supersede this stream, like in group_by/through/etc., you should use _chain() instead so *stream._next = cloned_stream* is set and *get_active_stream()* returns the cloned stream.

Return type *StreamT*[+T_co]

noack() → faust.types.streams.StreamT
Create new stream where acks are manual.

Return type *StreamT*[+T_co]

items() → AsyncIterator[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], T_co]]
Iterate over the stream as key, value pairs.

Examples

```
@app.agent(topic)
async def mytask(stream):
    async for key, value in stream.items():
        print(key, value)
```

Return type AsyncIterator[Tuple[Union[bytes, _ModelT, Any, None], +T_co]]

events() → AsyncIterable[faust.types.events.EventT]
Iterate over the stream as events exclusively.

This means the stream must be iterating over a channel, or at least an iterable of event objects.

Return type AsyncIterable[*EventT*[]]

take(*max_: int, within: Union[datetime.timedelta, float, str]*) → AsyncIterable[Sequence[T_co]]
Buffer n values at a time and yield a list of buffered values.

Parameters **within** (Union[timedelta, float, str]) – Timeout for when we give up waiting for another value, and process the values we have. Warning: If there's no timeout (i.e. *timeout=None*), the agent is likely to stall and block buffered events for an unreasonable length of time(!).

Return type AsyncIterable[Sequence[+T_co]]

enumerate(*start: int = 0*) → AsyncIterable[Tuple[int, T_co]]
Enumerate values received on this stream.

Unlike Python's built-in `enumerate`, this works with async generators.

Return type AsyncIterable[Tuple[int, +T_co]]

through(*channel: Union[str, faust.types.channels.ChannelT]*) → faust.types.streams.StreamT
Forward values to in this stream to channel.

Send messages received on this stream to another channel, and return a new stream that consumes from that channel.

Notes

The messages are forwarded after any processors have been applied.

Example

```
topic = app.topic('foo')

@app.agent(topic)
async def mytask(stream):
    async for value in stream.through(app.topic('bar')):
        # values were first received on topic 'foo'

        # then forwarded and consumed from topic 'bar'
        print(value)
```

Return type *StreamT*[+T_co]

echo(**channels: Union[str, faust.types.channels.ChannelT]*) → faust.types.streams.StreamT
Forward values to one or more channels.

Unlike *through()*, we don't consume from these channels.

Return type *StreamT*[+T_co]

group_by(*key: Union[faust.types.models.FieldDescriptorT, Callable[T, Union[bytes, faust.types.core._ModelT, Any, None]]], *, name: str = None, topic: faust.types.topics.TopicT = None, partitions: int = None*) → faust.types.streams.StreamT
Create new stream that repartitions the stream using a new key.

Parameters

- **key** (Union[*FieldDescriptorT*[~T], Callable[[~T], Union[bytes, _ModelT, Any, None]]]) – The key argument decides how the new key is generated, it can be a field descriptor, a callable, or an async callable.

  Note: The `name` argument must be provided if the key argument is a callable.

- **name** (Optional[str]) – Suffix to use for repartitioned topics. This argument is required if *key* is a callable.

Examples

Using a field descriptor to use a field in the event as the new key:

```
s = withdrawals_topic.stream()
# values in this stream are of type Withdrawal
async for event in s.group_by(Withdrawal.account_id):
    ...
```

Using an async callable to extract a new key:

```
s = withdrawals_topic.stream()

async def get_key(withdrawal):
    return await aiohttp.get(
        f'http://e.com/resolve_account/{withdrawal.account_id}')

async for event in s.group_by(get_key):
    ...
```

Using a regular callable to extract a new key:

```
s = withdrawals_topic.stream()

def get_key(withdrawal):
    return withdrawal.account_id.upper()

async for event in s.group_by(get_key):
    ...
```

Return type *StreamT*[+T_co]

filter(*fun: Callable[T]*) → faust.types.streams.StreamT
Filter values from stream using callback.

The callback may be a traditional function, lambda function, or an *async def* function.

This method is useful for filtering events before repartitioning a stream.

Examples

```
>>> async for v in stream.filter(lambda: v > 1000).group_by(...):
...     # do something
```

Return type *StreamT*[+T_co]

derive_topic(*name: str*, ***, *schema: faust.types.serializers.SchemaT = None*, *key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None*, *value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None*, *prefix: str = ''*, *suffix: str = ''*) → faust.types.topics.TopicT
Create Topic description derived from the K/V type of this stream.

Parameters

- **name** (str) – Topic name.

- **key_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – Specific key type to use for this topic. If not set, the key type of this stream will be used.
- **value_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – Specific value type to use for this topic. If not set, the value type of this stream will be used.

Raises **ValueError** – if the stream channel is not a topic.

Return type *TopicT*[]

async throw(*exc: BaseException*) → None
Send exception to stream iteration.

Return type None

combine(**nodes: faust.types.streams.JoinableT*, ***kwargs: Any*) → faust.types.streams.StreamT
Combine streams and tables into joined stream.

Return type *StreamT*[+T_co]

contribute_to_stream(*active: faust.types.streams.StreamT*) → None
Add stream as node in joined stream.

Return type None

async remove_from_stream(*stream: faust.types.streams.StreamT*) → None
Remove as node in a joined stream.

Return type None

join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Create stream where events are joined.

Return type *StreamT*[+T_co]

left_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Create stream where events are joined by LEFT JOIN.

Return type *StreamT*[+T_co]

inner_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Create stream where events are joined by INNER JOIN.

Return type *StreamT*[+T_co]

outer_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Create stream where events are joined by OUTER JOIN.

Return type *StreamT*[+T_co]

async on_merge(*value: T = None*) → Optional[T]
Signal called when an event is to be joined.

Return type Optional[~T]

async send(*value: T_contra*) → None
Send value into stream locally (bypasses topic).

Return type None

async on_start() → None
Signal called when the stream starts.

Return type None

async stop() → None
Stop this stream.

Return type None

async on_stop() → None
Signal that the stream is stopping.

Return type None

async ack(*event: faust.types.events.EventT*) → bool
Ack event.

This will decrease the reference count of the event message by one, and when the reference count reaches zero, the worker will commit the offset so that the message will not be seen by a worker again.

Parameters **event** (*EventT*[]) – Event to ack.

Return type bool

property label
Return description of stream, used in graphs and logs. :rtype: str

shortlabel
Return short description of stream.

faust.topics

Topic - Named channel using Kafka.

class faust.topics.**Topic**(*app: faust.types.app.AppT, *, topics: Sequence[str] = None, pattern: Union[str, Pattern[~AnyStr]] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, is_iterator: bool = False, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, replicas: int = None, acks: bool = True, internal: bool = False, config: Mapping[str, Any] = None, queue: mode.utils.queues.ThrowableQueue = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, maxsize: int = None, root: faust.types.channels.ChannelT = None, active_partitions: Set[faust.types.tuples.TP] = None, allow_empty: bool = None, has_prefix: bool = False, loop: asyncio.events.AbstractEventLoop = None*) → None
Define new topic description.

Parameters

- **app** (*AppT*[]) – App instance used to create this topic description.
- **topics** (Optional[Sequence[str]]) – List of topic names.
- **partitions** (Optional[int]) – Number of partitions for these topics. On declaration, topics are created using this. Note: If a message is produced before the topic is declared, and autoCreateTopics is enabled on the Kafka Server, the number of partitions used will be specified by the server configuration.
- **retention** (Union[timedelta, float, str, None]) – Number of seconds (as float/timedelta) to keep messages in the topic before they can be expired by the server.
- **pattern** (Union[str, Pattern[AnyStr], None]) – Regular expression evaluated to decide what topics to subscribe to. You cannot specify both topics and a pattern.

- **schema** (Optional[*SchemaT*[~KT, ~VT]]) – Schema used for serialization/deserialization.
- **key_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – How to deserialize keys for messages in this topic. Can be a `faust.Model` type, `str`, `bytes`, or `None` for "autodetect" (Overrides schema if one is defined).
- **value_type** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – How to deserialize values for messages in this topic. Can be a `faust.Model` type, `str`, `bytes`, or `None` for "autodetect" (Overrides schema if ones is defined).
- **active_partitions** (Optional[Set[*TP*]]) – Set of *faust.types.tuples.TP* that this topic should be restricted to.

Raises **TypeError** – if both *topics* and *pattern* is provided.

async send(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Send message to topic.

Return type Awaitable[*RecordMetadata*]

send_soon(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False, eager_partitioning: bool = False*) → faust.types.tuples.FutureMessage

Produce message by adding to buffer.

Notes

This method can be used by non-*async def* functions to produce messages.

Return type *FutureMessage*[]

async put(*event: faust.types.events.EventT*) → None

Put even directly onto the underlying queue of this topic.

This will only affect subscribers to a particular instance, in a particular process.

Return type None

property pattern

Regular expression used by this topic (if any). :rtype: Optional[Pattern[AnyStr]]

property partitions

Return the number of configured partitions for this topic.

Notes

This is only active for internal topics, fully owned and managed by Faust itself.

We never touch the configuration of a topic that exists in Kafka, and Kafka will sometimes automatically create topics when they don't exist. In this case the number of partitions for the automatically created topic will depend on the Kafka server configuration (`num.partitions`).

Always make sure your topics have the correct number of partitions. :rtype: `Optional`[`int`]

derive(***kwargs: Any*) → faust.types.channels.ChannelT
Create topic derived from the configuration of this topic.

Configuration will be copied from this topic, but any parameter overridden as a keyword argument.

See also:

derive_topic(): for a list of supported keyword arguments.

Return type *ChannelT*[]

derive_topic(**, topics: Sequence[str] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, internal: bool = None, config: Mapping[str, Any] = None, prefix: str = '', suffix: str = '', **kwargs: Any*) → faust.types.topics.TopicT
Create new topic with configuration derived from this topic.

Return type *TopicT*[]

get_topic_name() → str
Return the main topic name of this topic description.

As topic descriptions can have multiple topic names, this will only return when the topic has a singular topic name in the description.

Raises

- **TypeError** – if configured with a regular expression pattern.
- **ValueError** – if configured with multiple topic names.
- **TypeError** – if not configured with any names or patterns.

Return type `str`

async publish_message(*fut: faust.types.tuples.FutureMessage, wait: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]
Fulfill promise to publish message to topic.

Return type `Awaitable`[*RecordMetadata*]

maybe_declare
Declare/create this topic, only if it does not exist. :rtype: `None`

async declare() → None
Declare/create this topic on the server.

Return type `None`

faust.windows

Window Types.

class faust.windows.**Window**(**args*, ***kwargs*)
Base class for window types.

faust.windows.**HoppingWindow**
alias of faust.windows._PyHoppingWindow

class faust.windows.**TumblingWindow**(*size: Union[datetime.timedelta, float, str]*, *expires: Union[datetime.timedelta, float, str] = None*) → None
Tumbling window type.

Fixed-size, non-overlapping, gap-less windows.

faust.windows.**SlidingWindow**
alias of faust.windows._PySlidingWindow

faust.worker

Worker.

A "worker" starts a single instance of a Faust application.

See also:

Starting the App: for more information.

class faust.worker.**Worker**(*app: faust.types.app.AppT*, **services: mode.types.services.ServiceT*, *sensors: Iterable[faust.types.sensors.SensorT] = None*, *debug: bool = False*, *quiet: bool = False*, *loglevel: Union[str, int] = None*, *logfile: Union[str, IO] = None*, *stdout: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *stderr: IO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *blocking_timeout: float = 10.0*, *workdir: Union[pathlib.Path, str] = None*, *console_port: int = 50101*, *loop: asyncio.events.AbstractEventLoop = None*, *redirect_stdouts: bool = None*, *redirect_stdouts_level: int = None*, *logging_config: Dict = None*, ***kwargs: Any*) → None
Worker.

See also:

This is a subclass of mode.Worker.

Usage: You can start a worker using:

1) the **faust worker** program.

2) instantiating Worker programmatically and calling *execute_from_commandline()*:

```
>>> worker = Worker(app)
>>> worker.execute_from_commandline()
```

3) or if you already have an event loop, calling `await start`, but in that case *you are responsible for gracefully shutting down the event loop*:

```
async def start_worker(worker: Worker) -> None:
    await worker.start()
```

(continues on next page)

(continued from previous page)

```
def manage_loop():
    loop = asyncio.get_event_loop()
    worker = Worker(app, loop=loop)
    try:
        loop.run_until_complete(start_worker(worker)
    finally:
        worker.stop_and_shutdown_loop()
```

Parameters

- **app** (*AppT*[]) – The Faust app to start.
- ***services** – Services to start with worker. This includes application instances to start.
- **sensors** (*Iterable[SensorT]*) – List of sensors to include.
- **debug** (*bool*) – Enables debugging mode [disabled by default].
- **quiet** (*bool*) – Do not output anything to console [disabled by default].
- **loglevel** (*Union[str, int]*) – Level to use for logging, can be string (one of: CRIT|ERROR|WARN|INFO|DEBUG), or integer.
- **logfile** (*Union[str, IO]*) – Name of file or a stream to log to.
- **stdout** (*IO*) – Standard out stream.
- **stderr** (*IO*) – Standard err stream.
- **blocking_timeout** (*float*) – When `debug` is enabled this sets the timeout for detecting that the event loop is blocked.
- **workdir** (*Union[str, Path]*) – Custom working directory for the process that the worker will change into when started. This working directory change is permanent for the process, or until something else changes the working directory again.
- **loop** (*asyncio.AbstractEventLoop*) – Custom event loop object.

logger = <Logger faust.worker (WARNING)>

app = None
The Faust app started by this worker.

sensors = None
Additional sensors to add to the Faust app.

workdir = None
Current working directory. Note that if passed as an argument to Worker, the worker will change to this directory when started.

spinner = None
Class that displays a terminal progress spinner (see progress).

async on_start() → None
Signal called every time the worker starts.

Return type `None`

async on_startup_finished() → None
Signal called when worker has started.

Return type `None`

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return service dependencies that must start with the worker.

Return type Iterable[ServiceT[]]

async on_first_start() → None
Signal called the first time the worker starts.

First time, means this callback is not called if the worker is restarted by an exception being raised.

Return type None

change_workdir(*path: pathlib.Path*) → None
Change the current working directory (CWD).

Return type None

autodiscover() → None
Autodiscover modules and files to find @agent decorators, etc.

Return type None

async on_execute() → None
Signal called when the worker is about to start.

Return type None

on_worker_shutdown() → None
Signal called before the worker is shutting down.

Return type None

on_setup_root_logger(*logger: logging.Logger*, *level: int*) → None
Signal called when the root logger is being configured.

Return type None

1.6.2 App

faust.app

Application.

class faust.app.**App**(*id: str*, ***, *monitor: faust.sensors.monitor.Monitor = None*, *config_source: Any = None*, *loop: asyncio.events.AbstractEventLoop = None*, *beacon: mode.utils.types.trees.NodeT = None*, ***options: Any*) → None
Faust Application.

Parameters **id** (*str*) – Application ID.

Keyword Arguments **loop** (*asyncio.AbstractEventLoop*) – optional event loop to use.

See also:

Application Parameters – for supported keyword arguments.

SCAN_CATEGORIES = ['faust.agent', 'faust.command', 'faust.page', 'faust.service', 'fau

class BootStrategy(*app: faust.types.app.AppT*, ***, *enable_web: bool = None*, *enable_kafka: bool = None*, *enable_kafka_producer: bool = None*, *enable_kafka_consumer: bool = None*, *enable_sensors: bool = None*) → None
App startup strategy.

The startup strategy defines the graph of services to start when the Faust worker for an app starts.

agents() → Iterable[mode.types.services.ServiceT]
Return list of services required to start agents.
Return type Iterable[ServiceT[]]

client_only() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in client_only mode.
Return type Iterable[ServiceT[]]

enable_kafka = True

enable_kafka_consumer = None

enable_kafka_producer = None

enable_sensors = True

enable_web = None

kafka_client_consumer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka client consumer.
Return type Iterable[ServiceT[]]

kafka_conductor() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka conductor.
Return type Iterable[ServiceT[]]

kafka_consumer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka consumer.
Return type Iterable[ServiceT[]]

kafka_producer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka producer.
Return type Iterable[ServiceT[]]

producer_only() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in producer_only mode.
Return type Iterable[ServiceT[]]

sensors() → Iterable[mode.types.services.ServiceT]
Return list of services required to start sensors.
Return type Iterable[ServiceT[]]

server() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in default mode.
Return type Iterable[ServiceT[]]

tables() → Iterable[mode.types.services.ServiceT]
Return list of table-related services.
Return type Iterable[ServiceT[]]

web_components() → Iterable[mode.types.services.ServiceT]
Return list of web-related services (excluding web server).
Return type Iterable[ServiceT[]]

web_server() → Iterable[mode.types.services.ServiceT]
Return list of web-server services.
Return type Iterable[ServiceT[]]

class Settings(*id: str, *, debug: bool = None, version: int = None, broker: Union[str, yarl.URL, List[yarl.URL]] = None, broker_client_id: str = None, broker_request_timeout: Union[datetime.timedelta, float, str] = None, broker_credentials: Union[faust.types.auth.CredentialsT, ssl.SSLContext] = None, broker_commit_every: int = None, broker_commit_interval: Union[datetime.timedelta, float, str] = None, broker_commit_livelock_soft_timeout: Union[datetime.timedelta, float, str] = None, broker_session_timeout: Union[datetime.timedelta, float, str] = None, broker_heartbeat_interval: Union[datetime.timedelta, float, str] = None, broker_check_crcs: bool = None, broker_max_poll_records: int = None, broker_max_poll_interval: int = None, broker_consumer: Union[str, yarl.URL, List[yarl.URL]] = None, broker_producer: Union[str, yarl.URL, List[yarl.URL]] = None, agent_supervisor: Union[_T, str] = None, store: Union[str, yarl.URL] = None, cache: Union[str, yarl.URL] = None, web: Union[str, yarl.URL] = None, web_enabled: bool = True, processing_guarantee: Union[str, faust.types.enums.ProcessingGuarantee] = None, timezone: datetime.tzinfo = None, autodiscover: Union[bool, Iterable[str], Callable[Iterable[str]]] = None, origin: str = None, canonical_url: Union[str, yarl.URL] = None, datadir: Union[pathlib.Path, str] = None, tabledir: Union[pathlib.Path, str] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, logging_config: Dict = None, loghandlers: List[logging.Handler] = None, table_cleanup_interval: Union[datetime.timedelta, float, str] = None, table_standby_replicas: int = None, table_key_index_size: int = None, topic_replication_factor: int = None, topic_partitions: int = None, topic_allow_declare: bool = None, topic_disable_leader: bool = None, id_format: str = None, reply_to: str = None, reply_to_prefix: str = None, reply_create_topic: bool = None, reply_expires: Union[datetime.timedelta, float, str] = None, ssl_context: ssl.SSLContext = None, stream_buffer_maxsize: int = None, stream_wait_empty: bool = None, stream_ack_cancelled_tasks: bool = None, stream_ack_exceptions: bool = None, stream_publish_on_commit: bool = None, stream_recovery_delay: Union[datetime.timedelta, float, str] = None, producer_linger_ms: int = None, producer_max_batch_size: int = None, producer_acks: int = None, producer_max_request_size: int = None, producer_compression_type: str = None, producer_partitioner: Union[_T, str] = None, producer_request_timeout: Union[datetime.timedelta, float, str] = None, producer_api_version: str = None, consumer_max_fetch_size: int = None, consumer_auto_offset_reset: str = None, web_bind: str = None, web_port: int = None, web_host: str = None, web_transport: Union[str, yarl.URL] = None, web_in_thread: bool = None, web_cors_options: Mapping[str, faust.types.web.ResourceOptions] = None, worker_redirect_stdouts: bool = None, worker_redirect_stdouts_level: Union[int, str] = None, Agent: Union[_T, str] = None, ConsumerScheduler: Union[_T, str] = None, Event: Union[_T, str] = None, Schema: Union[_T, str] = None, Stream: Union[_T, str] = None, Table: Union[_T, str] = None, SetTable: Union[_T, str] = None, GlobalTable: Union[_T, str] = None, SetGlobalTable: Union[_T, str] = None, TableManager: Union[_T, str] = None, Serializers: Union[_T, str] = None, Worker: Union[_T, str] = None, PartitionAssignor: Union[_T, str] = None, LeaderAssignor: Union[_T, str] = None, Router: Union[_T, str] = None, Topic: Union[_T, str] = None, HttpClient: Union[_T, str] = None, Monitor: Union[_T, str] = None, url: Union[str, yarl.URL] = None, **kwargs: Any*) → None

property Agent

Return type Type[*AgentT*[]]

property ConsumerScheduler

Return type Type[SchedulingStrategyT]

property Event
Return type Type[*EventT*[]]

property GlobalTable
Return type Type[*GlobalTableT*[]]

property HttpClient
Return type Type[ClientSession]

property LeaderAssignor
Return type Type[*LeaderAssignorT*[]]

property Monitor
Return type Type[*SensorT*[]]

property PartitionAssignor
Return type Type[*PartitionAssignorT*]

property Router
Return type Type[*RouterT*]

property Schema
Return type Type[*SchemaT*[~KT, ~VT]]

property Serializers
Return type Type[*RegistryT*]

property SetGlobalTable
Return type Type[*GlobalTableT*[]]

property SetTable
Return type Type[*TableT*[~KT, ~VT]]

property Stream
Return type Type[*StreamT*[+T_co]]

property Table
Return type Type[*TableT*[~KT, ~VT]]

property TableManager
Return type Type[*TableManagerT*[]]

property Topic
Return type Type[*TopicT*[]]

property Worker
Return type Type[_WorkerT]

property agent_supervisor
Return type Type[SupervisorStrategyT]

property appdir
Return type Path

autodiscover = False

property broker
Return type List[URL]

broker_check_crcs = True

broker_client_id = 'faust-1.9.0'

broker_commit_every = 10000

property broker_commit_interval
Return type float

property broker_commit_livelock_soft_timeout
Return type float

property broker_consumer
Return type List[URL]

property broker_credentials
Return type Optional[*CredentialsT*]

property broker_heartbeat_interval
Return type float

broker_max_poll_interval = 1000.0

property broker_max_poll_records
Return type Optional[int]

property broker_producer
Return type List[URL]

property broker_request_timeout
Return type float

property broker_session_timeout
Return type float

property cache
Return type URL

property canonical_url
Return type URL

consumer_auto_offset_reset = 'earliest'

consumer_max_fetch_size = 4194304

property datadir
Return type Path

debug = False

find_old_versiondirs() → Iterable[pathlib.Path]
Return type Iterable[Path]

property id
Return type str

id_format = '{id}-v{self.version}'

key_serializer = 'raw'

logging_config = None

property name
Return type str

property origin
Return type Optional[str]

property processing_guarantee
Return type *ProcessingGuarantee*

producer_acks = -1

producer_api_version = 'auto'

producer_compression_type = None

producer_linger_ms = 0

producer_max_batch_size = 16384

producer_max_request_size = 1000000

property producer_partitioner

Return type Optional[Callable[[Optional[bytes], Sequence[int], Sequence[int]], int]]

property producer_request_timeout

Return type float

reply_create_topic = False

property reply_expires

Return type float

reply_to_prefix = 'f-reply-'

classmethod setting_names() → Set[str]

Return type Set[str]

ssl_context = None

property store

Return type URL

stream_ack_cancelled_tasks = True

stream_ack_exceptions = True

stream_buffer_maxsize = 4096

stream_publish_on_commit = False

property stream_recovery_delay

Return type float

stream_wait_empty = True

property table_cleanup_interval

Return type float

table_key_index_size = 1000

table_standby_replicas = 1

property tabledir

Return type Path

timezone = datetime.timezone.utc

topic_allow_declare = True

topic_disable_leader = False

topic_partitions = 8

topic_replication_factor = 1

value_serializer = 'json'

property version

Return type int

property web

Return type URL

web_bind = '0.0.0.0'

web_cors_options = None

web_host = 'build-10233069-project-230058-faust'

web_in_thread = False

web_port = 6066

property web_transport

Return type URL

worker_redirect_stdouts = True

worker_redirect_stdouts_level = 'WARN'

client_only = False

Set this to True if app should only start the services required to operate as an RPC client (producer and simple reply consumer).

producer_only = False

Set this to True if app should run without consumer/tables.

tracer = None

Optional tracing support.

on_init_dependencies() → Iterable[mode.types.services.ServiceT]

Return list of additional service dependencies.

The services returned will be started with the app when the app starts.

Return type Iterable[ServiceT[]]

async on_first_start() → None

Call first time app starts in this process.

Return type None

async on_start() → None

Call every time app start/restarts.

Return type None

async on_started() → None

Call when app is fully started.

Return type None

async on_started_init_extra_tasks() → None

Call when started to start additional tasks.

Return type None

async on_started_init_extra_services() → None

Call when initializing extra services at startup.

Return type None

async on_init_extra_service(*service: Union[mode.types.services.ServiceT, Type[mode.types.services.ServiceT]]*) → mode.types.services.ServiceT

Call when adding user services to this app.

Return type ServiceT[]

config_from_object(*obj: Any*, *, *silent: bool = False*, *force: bool = False*) → None

Read configuration from object.

Object is either an actual object or the name of a module to import.

Examples

```
>>> app.config_from_object('myproj.faustconfig')
```

```
>>> from myproj import faustconfig
>>> app.config_from_object(faustconfig)
```

Parameters

- **silent** (*bool*) – If true then import errors will be ignored.
- **force** (*bool*) – Force reading configuration immediately. By default the configuration will be read only when required.

Return type None

finalize() → None

Finalize app configuration.

Return type None

worker_init() → None

Init worker/CLI commands.

Return type None

worker_init_post_autodiscover() → None

Init worker after autodiscover.

Return type None

discover(**extra_modules: str, categories: Iterable[str] = None, ignore: Iterable[Any] = [<built-in method search of _sre.SRE_Pattern object>, '.__main__']*) → None

Discover decorators in packages.

Return type None

main() → NoReturn

Execute the **faust** umbrella command using this app.

Return type _NoReturn

topic(**topics: str, pattern: Union[str, Pattern[~AnyStr]] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, replicas: int = None, acks: bool = True, internal: bool = False, config: Mapping[str, Any] = None, maxsize: int = None, allow_empty: bool = False, has_prefix: bool = False, loop: asyncio.events.AbstractEventLoop = None*) → faust.types.topics.TopicT

Create topic description.

Topics are named channels (for example a Kafka topic), that exist on a server. To make an ephemeral local communication channel use: *channel()*.

See also:

faust.topics.Topic

Return type *TopicT*[]

channel(**, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, maxsize: int = None, loop: asyncio.events.AbstractEventLoop = None*) → faust.types.channels.ChannelT

Create new channel.

By default this will create an in-memory channel used for intra-process communication, but in practice channels can be backed by any transport (network or even means of inter-process communication).

See also:

faust.channels.Channel.

Return type *ChannelT*[]

agent(*channel: Union[str, faust.types.channels.ChannelT] = None, *, name: str = None, concurrency: int = 1, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, isolated_partitions: bool = False, use_reply_headers: bool = False, **kwargs: Any*) → Callable[Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], faust.types.agents.AgentT]

Create Agent from async def function.

It can be a regular async function:

```
@app.agent()
async def my_agent(stream):
    async for number in stream:
        print(f'Received: {number!r}')
```

Or it can be an async iterator that yields values. These values can be used as the reply in an RPC-style call, or for sinks: callbacks that forward events to other agents/topics/statsd, and so on:

```
@app.agent(sink=[log_topic])
async def my_agent(requests):
    async for number in requests:
        yield number * 2
```

Return type Callable[[Callable[[*StreamT*[+T_co]], Union[Coroutine[Any, Any, None], Awaitable[None], AsyncIterable[+T_co]]]], *AgentT*[]]

actor(*channel: Union[str, faust.types.channels.ChannelT] = None, *, name: str = None, concurrency: int = 1, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, isolated_partitions: bool = False, use_reply_headers: bool = False, **kwargs: Any*) → Callable[Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], faust.types.agents.AgentT]

Create Agent from async def function.

It can be a regular async function:

```
@app.agent()
async def my_agent(stream):
    async for number in stream:
        print(f'Received: {number!r}')
```

Or it can be an async iterator that yields values. These values can be used as the reply in an RPC-style call, or for sinks: callbacks that forward events to other agents/topics/statsd, and so on:

```
@app.agent(sink=[log_topic])
async def my_agent(requests):
    async for number in requests:
        yield number * 2
```

Return type Callable[[Callable[[*StreamT*[+T_co]], Union[Coroutine[Any, Any, None], Awaitable[None], AsyncIterable[+T_co]]]], *AgentT*[]]

task(*fun: Union[Callable[AppT, Awaitable], Callable[Awaitable]] = None, *, on_leader: bool = False, traced: bool = True*) → Union[Callable[Union[Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]], Union[Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]]], Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]]

Define an async def function to be started with the app.

This is like *timer()* but a one-shot task only executed at worker startup (after recovery and the worker is fully ready for operation).

The function may take zero, or one argument. If the target function takes an argument, the app argument is passed:

```
>>> @app.task
>>> async def on_startup(app):
...     print('STARTING UP: %r' % (app,))
```

Nullary functions are also supported:

```
>>> @app.task
>>> async def on_startup():
...     print('STARTING UP')
```

Return type Union[Callable[[Union[Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]], Union[Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]], Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]

timer(*interval: Union[datetime.timedelta, float, str], on_leader: bool = False, traced: bool = True, name: str = None, max_drift_correction: float = 0.1*) → Callable

Define an async def function to be run at periodic intervals.

Like *task()*, but executes periodically until the worker is shut down.

This decorator takes an async function and adds it to a list of timers started with the app.

Parameters

- **interval** (*Seconds*) – How often the timer executes in seconds.
- **on_leader** (*bool*) – Should the timer only run on the leader?

Example

```
>>> @app.timer(interval=10.0)
>>> async def every_10_seconds():
...     print('TEN SECONDS JUST PASSED')
```

```
>>> app.timer(interval=5.0, on_leader=True)
>>> async def every_5_seconds():
...     print('FIVE SECONDS JUST PASSED. ALSO, I AM THE LEADER!')
```

Return type Callable

crontab(*cron_format: str*, *, *timezone: datetime.tzinfo = None*, *on_leader: bool = False*, *traced: bool = True*) → Callable

Define periodic task using Crontab description.

This is an `async def` function to be run at the fixed times, defined by the Cron format.

Like *timer()*, but executes at fixed times instead of executing at certain intervals.

This decorator takes an async function and adds it to a list of Cronjobs started with the app.

Parameters **cron_format** (*str*) – The Cron spec defining fixed times to run the decorated function.

Keyword Arguments

- **timezone** – The timezone to be taken into account for the Cron jobs. If not set value from *timezone* will be taken.
- **on_leader** – Should the Cron job only run on the leader?

Example

```
>>> @app.crontab(cron_format='30 18 * * *',
                 timezone=pytz.timezone('US/Pacific'))
>>> async def every_6_30_pm_pacific():
...     print('IT IS 6:30pm')
```

```
>>> app.crontab(cron_format='30 18 * * *', on_leader=True)
>>> async def every_6_30_pm():
...     print('6:30pm UTC; ALSO, I AM THE LEADER!')
```

Return type Callable

service(*cls: Type[mode.types.services.ServiceT]*) → Type[mode.types.services.ServiceT]
Decorate `mode.Service` to be started with the app.

Examples

```
from mode import Service

@app.service
class Foo(Service):
    ...
```

Return type `Type`[`ServiceT`[]]

is_leader() → bool
Return `True` if we are in leader worker process.

Return type `bool`

stream(*channel: Union[AsyncIterable, Iterable], beacon: mode.utils.types.trees.NodeT = None, **kwargs: Any*) → faust.types.streams.StreamT
Create new stream from channel/topic/iterable/async iterable.

Parameters

- **channel** (`Union`[`AsyncIterable`[+T_co], `Iterable`[+T_co]]) – Iterable to stream over (async or non-async).
- **kwargs** (`Any`) – See `Stream`.

Return type *`StreamT`*[+T_co]

Returns to iterate over events in the stream.

Return type *faust.Stream*

Table(*name: str*, *, *default: Callable[Any] = None*, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *help: str = None*, **kwargs: Any*) → faust.types.tables.TableT
Define new table.

Parameters

- **name** (`str`) – Name used for table, note that two tables living in the same application cannot have the same name.
- **default** (`Optional`[`Callable`[[], `Any`]]) – A callable, or type that will return a default value for keys missing in this table.
- **window** (`Optional`[*`WindowT`*]) – A windowing strategy to wrap this window in.

Examples

```
>>> table = app.Table('user_to_amount', default=int)
>>> table['George']
0
>>> table['Elaine'] += 1
>>> table['Elaine'] += 1
>>> table['Elaine']
2
```

Return type *TableT*[~KT, ~VT]

GlobalTable(*name: str*, ***, *default: Callable[Any] = None*, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.GlobalTableT

Define new global table.

Parameters

- **name** (*str*) – Name used for global table, note that two global tables living in the same application cannot have the same name.
- **default** (*Optional*[*Callable*[[], *Any*]]) – A callable, or type that will return a default valu for keys missing in this global table.
- **window** (*Optional*[*WindowT*]) – A windowing strategy to wrap this window in.

Examples

```
>>> gtable = app.GlobalTable('user_to_amount', default=int)
>>> gtable['George']
0
>>> gtable['Elaine'] += 1
>>> gtable['Elaine'] += 1
>>> gtable['Elaine']
2
```

Return type *GlobalTableT*[]

SetTable(*name: str*, ***, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *start_manager: bool = False*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.TableT

Table of sets.

Return type *TableT*[~KT, ~VT]

SetGlobalTable(*name: str*, ***, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *start_manager: bool = False*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.TableT

Table of sets (global).

Return type *TableT*[~KT, ~VT]

page(*path: str*, ***, *base: Type[faust.web.views.View] = <class 'faust.web.views.View'>*, *cors_options: Mapping[str, faust.types.web.ResourceOptions] = None*, *name: str = None*) → Callable[Union[Type[faust.types.web.View], Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Type[faust.web.views.View]]

Decorate view to be included in the web server.

Return type *Callable*[[*Union*[*Type*[*View*], *Callable*[[*View*, *Request*], *Union*[*Coroutine*[*Any*, *Any*, *Response*], *Awaitable*[*Response*]]], *Callable*[[*View*, *Request*, *Any*, *Any*], *Union*[*Coroutine*[*Any*, *Any*, *Response*], *Awaitable*[*Response*]]]]], *Type*[*View*]]

table_route(*table: faust.types.tables.CollectionT*, *shard_param: str = None*, ***, *query_param: str = None*, *match_info: str = None*, *exact_key: str = None*) → Callable[Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]]

Decorate view method to route request to table key destination.

Return type Callable[[Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]], Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]]

command(**options: Any*, *base: Optional[Type[faust.app.base._AppCommand]] = None*, ***kwargs: Any*) → Callable[Callable, Type[faust.app.base._AppCommand]]

Decorate `async def` function to be used as CLI command.

Return type Callable[[Callable], Type[_AppCommand]]

create_event(*key: Union[bytes, faust.types.core._ModelT, Any, None]*, *value: Union[bytes, faust.types.core._ModelT, Any]*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]*, *message: faust.types.tuples.Message*) → faust.types.events.EventT

Create new *faust.Event* object.

Return type *EventT*[]

async start_client() → None

Start the app in Client-Only mode necessary for RPC requests.

Notes

Once started as a client the app cannot be restarted as Server.

Return type None

async maybe_start_client() → None

Start the app in Client-Only mode if not started as Server.

Return type None

trace(*name: str*, *trace_enabled: bool = True*, ***extra_context: Any*) → ContextManager

Return new trace context to trace operation using OpenTracing.

Return type ContextManager[+T_co]

traced(*fun: Callable*, *name: str = None*, *sample_rate: float = 1.0*, ***context: Any*) → Callable

Decorate function to be traced using the OpenTracing API.

Return type Callable

async send(*channel: Union[faust.types.channels.ChannelT, str], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*) → Awaitable[faust.types.tuples.RecordMetadata]

Send event to channel/topic.

Parameters

- **channel** (Union[*ChannelT*[], str]) – Channel/topic or the name of a topic to send event to.
- **key** (Union[bytes, _ModelT, Any, None]) – Message key.
- **value** (Union[bytes, _ModelT, Any, None]) – Message value.
- **partition** (Optional[int]) – Specific partition to send to. If not set the partition will be chosen by the partitioner.
- **timestamp** (Optional[float]) – Epoch seconds (from Jan 1 1970 UTC) to use as the message timestamp. Defaults to current time.
- **headers** (Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]) – Mapping of key/value pairs, or iterable of key value pairs to use as headers for the message.
- **schema** (Optional[*SchemaT*[~KT, ~VT]]) – *Schema* to use for serialization.
- **key_serializer** (Union[*CodecT*, str, None]) – Serializer to use (if value is not model). Overrides schema if one is specified.
- **value_serializer** (Union[*CodecT*, str, None]) – Serializer to use (if value is not model). Overrides schema if one is specified.
- **callback** (Optional[Callable[[*FutureMessage*[]], Union[None, Awaitable[None]]]]) – Called after the message is fully delivered to the channel, but not to the consumer. Signature must be unary as the *FutureMessage* future is passed to it.

  The resulting *faust.types.tuples.RecordMetadata* object is then available as `fut.result()`.

Return type Awaitable[*RecordMetadata*]

in_transaction

Return `True` if stream is using transactions.

LiveCheck(***kwargs: Any*) → faust.app.base._LiveCheck

Return new LiveCheck instance testing features for this app.

Return type _LiveCheck

maybe_start_producer

Ensure producer is started. :rtype: *ProducerT*[]

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]]*) → bool

Commit offset for acked messages in specified topics'.

> **Warning:** This will commit acked messages in **all topics** if the topics argument is passed in as `None`.

Return type bool

async on_stop() → None
Call when application stops.

Tip: Remember to call `super` if you override this method.

Return type None

on_rebalance_start() → None
Call when rebalancing starts.

Return type None

on_rebalance_return() → None

Return type None

on_rebalance_end() → None
Call when rebalancing is done.

Return type None

FlowControlQueue(*maxsize: int = None*, ***, *clear_on_resume: bool = False*, *loop: asyncio.events.AbstractEventLoop = None*) → mode.utils.queues.ThrowableQueue
Like `asyncio.Queue`, but can be suspended/resumed.

Return type ThrowableQueue

Worker(***kwargs: Any*) → faust.app.base._Worker
Return application worker instance.

Return type _Worker

on_webserver_init(*web: faust.types.web.Web*) → None
Call when the Web server is initializing.

Return type None

property conf
Application configuration. :rtype: *Settings*

property producer
Message producer. :rtype: *ProducerT*[]

property consumer
Message consumer. :rtype: *ConsumerT*[]

property transport
Consumer message transport. :rtype: *TransportT*

logger = <Logger faust.app.base (WARNING)>

property producer_transport
Producer message transport. :rtype: *TransportT*

property cache
Cache backend. :rtype: *CacheBackendT*[]

tables
Map of available tables, and the table manager service.

topics
Topic Conductor.

This is the mediator that moves messages fetched by the Consumer into the streams.

It's also a set of registered topics by string topic name, so you can check if a topic is being consumed from by doing `topic in app.topics`.

property monitor
Monitor keeps stats about what's going on inside the worker. :rtype: *Monitor*[]

flow_control
Flow control of streams.

This object controls flow into stream queues, and can also clear all buffers.

property http_client
HTTP client Session. :rtype: `ClientSession`

assignor
Partition Assignor.

Responsible for partition assignment.

router
Find the node partitioned data belongs to.

The router helps us route web requests to the wanted Faust node. If a topic is sharded by account_id, the router can send us to the Faust worker responsible for any account. Used by the `@app.table_route` decorator.

web
Web driver.

serializers
Return serializer registry.

property label
Return human readable description of application. :rtype: `str`

property shortlabel
Return short description of application. :rtype: `str`

class faust.app.**BootStrategy**(*app: faust.types.app.AppT*, ***, *enable_web: bool = None*, *enable_kafka: bool = None*, *enable_kafka_producer: bool = None*, *enable_kafka_consumer: bool = None*, *enable_sensors: bool = None*) → None

App startup strategy.

The startup strategy defines the graph of services to start when the Faust worker for an app starts.

enable_kafka = True

enable_kafka_producer = None

enable_kafka_consumer = None

enable_web = None

enable_sensors = True

server() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in default mode.

Return type `Iterable`[`ServiceT`[]]

client_only() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in client_only mode.

Return type Iterable[ServiceT[]]

producer_only() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in producer_only mode.

Return type Iterable[ServiceT[]]

sensors() → Iterable[mode.types.services.ServiceT]
Return list of services required to start sensors.

Return type Iterable[ServiceT[]]

kafka_producer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka producer.

Return type Iterable[ServiceT[]]

kafka_consumer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka consumer.

Return type Iterable[ServiceT[]]

kafka_client_consumer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka client consumer.

Return type Iterable[ServiceT[]]

agents() → Iterable[mode.types.services.ServiceT]
Return list of services required to start agents.

Return type Iterable[ServiceT[]]

kafka_conductor() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka conductor.

Return type Iterable[ServiceT[]]

web_server() → Iterable[mode.types.services.ServiceT]
Return list of web-server services.

Return type Iterable[ServiceT[]]

web_components() → Iterable[mode.types.services.ServiceT]
Return list of web-related services (excluding web server).

Return type Iterable[ServiceT[]]

tables() → Iterable[mode.types.services.ServiceT]
Return list of table-related services.

Return type Iterable[ServiceT[]]

faust.app.base

Faust Application.

An app is an instance of the Faust library. Everything starts here.

class faust.app.base.**BootStrategy**(*app: faust.types.app.AppT*, ***, *enable_web: bool = None*, *enable_kafka: bool = None*, *enable_kafka_producer: bool = None*, *enable_kafka_consumer: bool = None*, *enable_sensors: bool = None*) → None
App startup strategy.

The startup strategy defines the graph of services to start when the Faust worker for an app starts.

enable_kafka = True

enable_kafka_producer = None

enable_kafka_consumer = None

enable_web = None

enable_sensors = True

server() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in default mode.

Return type Iterable[ServiceT[]]

client_only() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in client_only mode.

Return type Iterable[ServiceT[]]

producer_only() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in producer_only mode.

Return type Iterable[ServiceT[]]

sensors() → Iterable[mode.types.services.ServiceT]
Return list of services required to start sensors.

Return type Iterable[ServiceT[]]

kafka_producer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka producer.

Return type Iterable[ServiceT[]]

kafka_consumer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka consumer.

Return type Iterable[ServiceT[]]

kafka_client_consumer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka client consumer.

Return type Iterable[ServiceT[]]

agents() → Iterable[mode.types.services.ServiceT]
Return list of services required to start agents.

Return type Iterable[ServiceT[]]

kafka_conductor() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka conductor.

Return type Iterable[ServiceT[]]

web_server() → Iterable[mode.types.services.ServiceT]
Return list of web-server services.

Return type Iterable[ServiceT[]]

web_components() → Iterable[mode.types.services.ServiceT]
Return list of web-related services (excluding web server).

Return type Iterable[ServiceT[]]

tables() → Iterable[mode.types.services.ServiceT]
Return list of table-related services.

Return type Iterable[ServiceT[]]

class faust.app.base.**App**(*id: str*, ***, *monitor: faust.sensors.monitor.Monitor = None*, *config_source: Any = None*, *loop: asyncio.events.AbstractEventLoop = None*, *beacon: mode.utils.types.trees.NodeT = None*, ***options: Any*) → None

Faust Application.

Parameters **id** (*str*) – Application ID.

Keyword Arguments **loop** (*asyncio.AbstractEventLoop*) – optional event loop to use.

See also:

Application Parameters – for supported keyword arguments.

SCAN_CATEGORIES = ['faust.agent', 'faust.command', 'faust.page', 'faust.service', 'fau

class BootStrategy(*app: faust.types.app.AppT*, ***, *enable_web: bool = None*, *enable_kafka: bool = None*, *enable_kafka_producer: bool = None*, *enable_kafka_consumer: bool = None*, *enable_sensors: bool = None*) → None

App startup strategy.

The startup strategy defines the graph of services to start when the Faust worker for an app starts.

agents() → Iterable[mode.types.services.ServiceT]
Return list of services required to start agents.
Return type Iterable[ServiceT[]]

client_only() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in client_only mode.
Return type Iterable[ServiceT[]]

enable_kafka = True

enable_kafka_consumer = None

enable_kafka_producer = None

enable_sensors = True

enable_web = None

kafka_client_consumer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka client consumer.
Return type Iterable[ServiceT[]]

kafka_conductor() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka conductor.
Return type Iterable[ServiceT[]]

kafka_consumer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka consumer.
Return type Iterable[ServiceT[]]

kafka_producer() → Iterable[mode.types.services.ServiceT]
Return list of services required to start Kafka producer.
Return type Iterable[ServiceT[]]

producer_only() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in producer_only mode.
Return type Iterable[ServiceT[]]

sensors() → Iterable[mode.types.services.ServiceT]
Return list of services required to start sensors.
Return type Iterable[ServiceT[]]

server() → Iterable[mode.types.services.ServiceT]
Return services to start when app is in default mode.
Return type Iterable[ServiceT[]]

tables() → Iterable[mode.types.services.ServiceT]
Return list of table-related services.
Return type Iterable[ServiceT[]]

web_components() → Iterable[mode.types.services.ServiceT]
Return list of web-related services (excluding web server).
Return type Iterable[ServiceT[]]

web_server() → Iterable[mode.types.services.ServiceT]
Return list of web-server services.
Return type Iterable[ServiceT[]]

class Settings(*id: str, *, debug: bool = None, version: int = None, broker: Union[str, yarl.URL, List[yarl.URL]] = None, broker_client_id: str = None, broker_request_timeout: Union[datetime.timedelta, float, str] = None, broker_credentials: Union[faust.types.auth.CredentialsT, ssl.SSLContext] = None, broker_commit_every: int = None, broker_commit_interval: Union[datetime.timedelta, float, str] = None, broker_commit_livelock_soft_timeout: Union[datetime.timedelta, float, str] = None, broker_session_timeout: Union[datetime.timedelta, float, str] = None, broker_heartbeat_interval: Union[datetime.timedelta, float, str] = None, broker_check_crcs: bool = None, broker_max_poll_records: int = None, broker_max_poll_interval: int = None, broker_consumer: Union[str, yarl.URL, List[yarl.URL]] = None, broker_producer: Union[str, yarl.URL, List[yarl.URL]] = None, agent_supervisor: Union[_T, str] = None, store: Union[str, yarl.URL] = None, cache: Union[str, yarl.URL] = None, web: Union[str, yarl.URL] = None, web_enabled: bool = True, processing_guarantee: Union[str, faust.types.enums.ProcessingGuarantee] = None, timezone: datetime.tzinfo = None, autodiscover: Union[bool, Iterable[str], Callable[Iterable[str]]] = None, origin: str = None, canonical_url: Union[str, yarl.URL] = None, datadir: Union[pathlib.Path, str] = None, tabledir: Union[pathlib.Path, str] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, logging_config: Dict = None, loghandlers: List[logging.Handler] = None, table_cleanup_interval: Union[datetime.timedelta, float, str] = None, table_standby_replicas: int = None, table_key_index_size: int = None, topic_replication_factor: int = None, topic_partitions: int = None, topic_allow_declare: bool = None, topic_disable_leader: bool = None, id_format: str = None, reply_to: str = None, reply_to_prefix: str = None, reply_create_topic: bool = None, reply_expires: Union[datetime.timedelta, float, str] = None, ssl_context: ssl.SSLContext = None, stream_buffer_maxsize: int = None, stream_wait_empty: bool = None, stream_ack_cancelled_tasks: bool = None, stream_ack_exceptions: bool = None, stream_publish_on_commit: bool = None, stream_recovery_delay: Union[datetime.timedelta, float, str] = None, producer_linger_ms: int = None, producer_max_batch_size: int = None, producer_acks: int = None, producer_max_request_size: int = None, producer_compression_type: str = None, producer_partitioner: Union[_T, str] = None, producer_request_timeout: Union[datetime.timedelta, float, str] = None, producer_api_version: str = None, consumer_max_fetch_size: int = None, consumer_auto_offset_reset: str = None, web_bind: str = None, web_port: int = None, web_host: str = None, web_transport: Union[str, yarl.URL] = None, web_in_thread: bool = None, web_cors_options: Mapping[str, faust.types.web.ResourceOptions] = None, worker_redirect_stdouts: bool = None, worker_redirect_stdouts_level: Union[int, str] = None, Agent: Union[_T, str] = None, ConsumerScheduler: Union[_T, str] = None, Event: Union[_T, str] = None, Schema: Union[_T, str] = None, Stream: Union[_T, str] = None, Table: Union[_T, str] = None, SetTable: Union[_T, str] = None, GlobalTable: Union[_T, str] = None, SetGlobalTable: Union[_T, str] = None, TableManager: Union[_T, str] = None, Serializers: Union[_T, str] = None, Worker: Union[_T, str] = None, PartitionAssignor: Union[_T, str] = None, LeaderAssignor: Union[_T, str] = None, Router: Union[_T, str] = None, Topic: Union[_T, str] = None, HttpClient: Union[_T, str] = None, Monitor: Union[_T, str] = None, url: Union[str, yarl.URL] = None, **kwargs: Any*) → None

property Agent

Return type Type[*AgentT*[]]

property ConsumerScheduler

Return type Type[SchedulingStrategyT]

property Event
Return type Type[*EventT*[]]

property GlobalTable
Return type Type[*GlobalTableT*[]]

property HttpClient
Return type Type[ClientSession]

property LeaderAssignor
Return type Type[*LeaderAssignorT*[]]

property Monitor
Return type Type[*SensorT*[]]

property PartitionAssignor
Return type Type[*PartitionAssignorT*]

property Router
Return type Type[*RouterT*]

property Schema
Return type Type[*SchemaT*[~KT, ~VT]]

property Serializers
Return type Type[*RegistryT*]

property SetGlobalTable
Return type Type[*GlobalTableT*[]]

property SetTable
Return type Type[*TableT*[~KT, ~VT]]

property Stream
Return type Type[*StreamT*[+T_co]]

property Table
Return type Type[*TableT*[~KT, ~VT]]

property TableManager
Return type Type[*TableManagerT*[]]

property Topic
Return type Type[*TopicT*[]]

property Worker
Return type Type[_WorkerT]

property agent_supervisor
Return type Type[SupervisorStrategyT]

property appdir
Return type Path

autodiscover = False

property broker
Return type List[URL]

broker_check_crcs = True

broker_client_id = 'faust-1.9.0'

broker_commit_every = 10000

property broker_commit_interval
Return type float

property broker_commit_livelock_soft_timeout
Return type float

property broker_consumer
Return type List[URL]

property broker_credentials
Return type Optional[*CredentialsT*]

property broker_heartbeat_interval
Return type float

broker_max_poll_interval = 1000.0

property broker_max_poll_records
Return type Optional[int]

property broker_producer
Return type List[URL]

property broker_request_timeout
Return type float

property broker_session_timeout
Return type float

property cache
Return type URL

property canonical_url
Return type URL

consumer_auto_offset_reset = 'earliest'

consumer_max_fetch_size = 4194304

property datadir
Return type Path

debug = False

find_old_versiondirs() → Iterable[pathlib.Path]
Return type Iterable[Path]

property id
Return type str

id_format = '{id}-v{self.version}'

key_serializer = 'raw'

logging_config = None

property name
Return type str

property origin
Return type Optional[str]

property processing_guarantee
Return type *ProcessingGuarantee*

producer_acks = -1

producer_api_version = 'auto'

producer_compression_type = None

producer_linger_ms = 0

producer_max_batch_size = 16384

producer_max_request_size = 1000000

property producer_partitioner

Return type Optional[Callable[[Optional[bytes], Sequence[int], Sequence[int]], int]]

property producer_request_timeout

Return type float

reply_create_topic = False

property reply_expires

Return type float

reply_to_prefix = 'f-reply-'

classmethod setting_names() → Set[str]

Return type Set[str]

ssl_context = None

property store

Return type URL

stream_ack_cancelled_tasks = True

stream_ack_exceptions = True

stream_buffer_maxsize = 4096

stream_publish_on_commit = False

property stream_recovery_delay

Return type float

stream_wait_empty = True

property table_cleanup_interval

Return type float

table_key_index_size = 1000

table_standby_replicas = 1

property tabledir

Return type Path

timezone = datetime.timezone.utc

topic_allow_declare = True

topic_disable_leader = False

topic_partitions = 8

topic_replication_factor = 1

value_serializer = 'json'

property version

Return type int

property web
Return type URL

web_bind = '0.0.0.0'

web_cors_options = None

web_host = 'build-10233069-project-230058-faust'

web_in_thread = False

web_port = 6066

property web_transport
Return type URL

worker_redirect_stdouts = True

worker_redirect_stdouts_level = 'WARN'

client_only = False
Set this to True if app should only start the services required to operate as an RPC client (producer and simple reply consumer).

producer_only = False
Set this to True if app should run without consumer/tables.

tracer = None
Optional tracing support.

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of additional service dependencies.

The services returned will be started with the app when the app starts.

Return type Iterable[ServiceT[]]

async on_first_start() → None
Call first time app starts in this process.

Return type None

async on_start() → None
Call every time app start/restarts.

Return type None

async on_started() → None
Call when app is fully started.

Return type None

async on_started_init_extra_tasks() → None
Call when started to start additional tasks.

Return type None

async on_started_init_extra_services() → None
Call when initializing extra services at startup.

Return type None

async on_init_extra_service(*service: Union[mode.types.services.ServiceT, Type[mode.types.services.ServiceT]]*) → mode.types.services.ServiceT

Call when adding user services to this app.

Return type ServiceT[]

config_from_object(*obj: Any*, ***, *silent: bool = False*, *force: bool = False*) → None

Read configuration from object.

Object is either an actual object or the name of a module to import.

Examples

```
>>> app.config_from_object('myproj.faustconfig')
```

```
>>> from myproj import faustconfig
>>> app.config_from_object(faustconfig)
```

Parameters

- **silent** (*bool*) – If true then import errors will be ignored.
- **force** (*bool*) – Force reading configuration immediately. By default the configuration will be read only when required.

Return type None

finalize() → None

Finalize app configuration.

Return type None

worker_init() → None

Init worker/CLI commands.

Return type None

worker_init_post_autodiscover() → None

Init worker after autodiscover.

Return type None

discover(**extra_modules: str*, *categories: Iterable[str] = None*, *ignore: Iterable[Any] = [<built-in method search of _sre.SRE_Pattern object>, '.__main__']*) → None

Discover decorators in packages.

Return type None

main() → NoReturn

Execute the **faust** umbrella command using this app.

Return type _NoReturn

topic(**topics: str, pattern: Union[str, Pattern[~AnyStr]] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, replicas: int = None, acks: bool = True, internal: bool = False, config: Mapping[str, Any] = None, maxsize: int = None, allow_empty: bool = False, has_prefix: bool = False, loop: asyncio.events.AbstractEventLoop = None*) → faust.types.topics.TopicT

Create topic description.

Topics are named channels (for example a Kafka topic), that exist on a server. To make an ephemeral local communication channel use: *channel()*.

See also:

faust.topics.Topic

Return type *TopicT*[]

channel(**, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, maxsize: int = None, loop: asyncio.events.AbstractEventLoop = None*) → faust.types.channels.ChannelT

Create new channel.

By default this will create an in-memory channel used for intra-process communication, but in practice channels can be backed by any transport (network or even means of inter-process communication).

See also:

faust.channels.Channel.

Return type *ChannelT*[]

agent(*channel: Union[str, faust.types.channels.ChannelT] = None, *, name: str = None, concurrency: int = 1, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, isolated_partitions: bool = False, use_reply_headers: bool = False, **kwargs: Any*) → Callable[Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], faust.types.agents.AgentT]

Create Agent from async def function.

It can be a regular async function:

```
@app.agent()
async def my_agent(stream):
    async for number in stream:
        print(f'Received: {number!r}')
```

Or it can be an async iterator that yields values. These values can be used as the reply in an RPC-style call, or for sinks: callbacks that forward events to other agents/topics/statsd, and so on:

```
@app.agent(sink=[log_topic])
async def my_agent(requests):
    async for number in requests:
        yield number * 2
```

Return type Callable[[Callable[[*StreamT*[+T_co]], Union[Coroutine[Any, Any, None], Awaitable[None], AsyncIterable[+T_co]]]], *AgentT*[]]

actor(*channel: Union[str, faust.types.channels.ChannelT] = None, *, name: str = None, concurrency: int = 1, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, isolated_partitions: bool = False, use_reply_headers: bool = False, **kwargs: Any*) → Callable[Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], faust.types.agents.AgentT]

Create Agent from async def function.

It can be a regular async function:

```
@app.agent()
async def my_agent(stream):
    async for number in stream:
        print(f'Received: {number!r}')
```

Or it can be an async iterator that yields values. These values can be used as the reply in an RPC-style call, or for sinks: callbacks that forward events to other agents/topics/statsd, and so on:

```
@app.agent(sink=[log_topic])
async def my_agent(requests):
    async for number in requests:
        yield number * 2
```

Return type Callable[[Callable[[*StreamT*[+T_co]], Union[Coroutine[Any, Any, None], Awaitable[None], AsyncIterable[+T_co]]]], *AgentT*[]]

task(*fun: Union[Callable[AppT, Awaitable], Callable[Awaitable]] = None, *, on_leader: bool = False, traced: bool = True*) → Union[Callable[Union[Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]], Union[Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]]], Callable[faust.types.app.AppT, Awaitable], Callable[Awaitable]]

Define an async def function to be started with the app.

This is like *timer()* but a one-shot task only executed at worker startup (after recovery and the worker is fully ready for operation).

The function may take zero, or one argument. If the target function takes an argument, the app argument is passed:

```
>>> @app.task
>>> async def on_startup(app):
...     print('STARTING UP: %r' % (app,))
```

Nullary functions are also supported:

```
>>> @app.task
>>> async def on_startup():
...     print('STARTING UP')
```

Return type Union[Callable[[Union[Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]], Union[Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]], Callable[[*AppT*[]], Awaitable[+T_co]], Callable[[], Awaitable[+T_co]]]

timer(*interval: Union[datetime.timedelta, float, str], on_leader: bool = False, traced: bool = True, name: str = None, max_drift_correction: float = 0.1*) → Callable

Define an async def function to be run at periodic intervals.

Like *task()*, but executes periodically until the worker is shut down.

This decorator takes an async function and adds it to a list of timers started with the app.

Parameters

- **interval** (*Seconds*) – How often the timer executes in seconds.
- **on_leader** (*bool*) – Should the timer only run on the leader?

Example

```
>>> @app.timer(interval=10.0)
>>> async def every_10_seconds():
...     print('TEN SECONDS JUST PASSED')
```

```
>>> app.timer(interval=5.0, on_leader=True)
>>> async def every_5_seconds():
...     print('FIVE SECONDS JUST PASSED. ALSO, I AM THE LEADER!')
```

Return type Callable

crontab(*cron_format: str*, *, *timezone: datetime.tzinfo = None*, *on_leader: bool = False*, *traced: bool = True*) → Callable

Define periodic task using Crontab description.

This is an `async def` function to be run at the fixed times, defined by the Cron format.

Like *timer()*, but executes at fixed times instead of executing at certain intervals.

This decorator takes an async function and adds it to a list of Cronjobs started with the app.

Parameters **cron_format** (*str*) – The Cron spec defining fixed times to run the decorated function.

Keyword Arguments

- **timezone** – The timezone to be taken into account for the Cron jobs. If not set value from *timezone* will be taken.
- **on_leader** – Should the Cron job only run on the leader?

Example

```
>>> @app.crontab(cron_format='30 18 * * *',
                 timezone=pytz.timezone('US/Pacific'))
>>> async def every_6_30_pm_pacific():
...     print('IT IS 6:30pm')
```

```
>>> app.crontab(cron_format='30 18 * * *', on_leader=True)
>>> async def every_6_30_pm():
...     print('6:30pm UTC; ALSO, I AM THE LEADER!')
```

Return type Callable

service(*cls: Type[mode.types.services.ServiceT]*) → Type[mode.types.services.ServiceT]
Decorate `mode.Service` to be started with the app.

Examples

```
from mode import Service

@app.service
class Foo(Service):
    ...
```

Return type `Type`[`ServiceT`[]]

is_leader() → bool
Return `True` if we are in leader worker process.

Return type `bool`

stream(*channel: Union[AsyncIterable, Iterable], beacon: mode.utils.types.trees.NodeT = None, **kwargs: Any*) → faust.types.streams.StreamT
Create new stream from channel/topic/iterable/async iterable.

Parameters

- **channel** (`Union`[`AsyncIterable`[+T_co], `Iterable`[+T_co]]) – Iterable to stream over (async or non-async).
- **kwargs** (`Any`) – See `Stream`.

Return type *`StreamT`*[+T_co]

Returns to iterate over events in the stream.

Return type *faust.Stream*

Table(*name: str*, *, *default: Callable[Any] = None*, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *help: str = None*, **kwargs: Any*) → faust.types.tables.TableT
Define new table.

Parameters

- **name** (`str`) – Name used for table, note that two tables living in the same application cannot have the same name.
- **default** (`Optional`[`Callable`[[], `Any`]]) – A callable, or type that will return a default value for keys missing in this table.
- **window** (`Optional`[*`WindowT`*]) – A windowing strategy to wrap this window in.

Examples

```
>>> table = app.Table('user_to_amount', default=int)
>>> table['George']
0
>>> table['Elaine'] += 1
>>> table['Elaine'] += 1
>>> table['Elaine']
2
```

Return type *TableT*[~KT, ~VT]

GlobalTable(*name: str*, ***, *default: Callable[Any] = None*, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.GlobalTableT

Define new global table.

Parameters

- **name** (*str*) – Name used for global table, note that two global tables living in the same application cannot have the same name.
- **default** (*Optional*[*Callable*[[], *Any*]]) – A callable, or type that will return a default valu for keys missing in this global table.
- **window** (*Optional*[*WindowT*]) – A windowing strategy to wrap this window in.

Examples

```
>>> gtable = app.GlobalTable('user_to_amount', default=int)
>>> gtable['George']
0
>>> gtable['Elaine'] += 1
>>> gtable['Elaine'] += 1
>>> gtable['Elaine']
2
```

Return type *GlobalTableT*[]

SetTable(*name: str*, ***, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *start_manager: bool = False*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.TableT

Table of sets.

Return type *TableT*[~KT, ~VT]

SetGlobalTable(*name: str*, ***, *window: faust.types.windows.WindowT = None*, *partitions: int = None*, *start_manager: bool = False*, *help: str = None*, ***kwargs: Any*) → faust.types.tables.TableT

Table of sets (global).

Return type *TableT*[~KT, ~VT]

page(*path: str*, ***, *base: Type[faust.web.views.View] = <class 'faust.web.views.View'>*, *cors_options: Mapping[str, faust.types.web.ResourceOptions] = None*, *name: str = None*) → Callable[Union[Type[faust.types.web.View], Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Type[faust.web.views.View]]

Decorate view to be included in the web server.

Return type *Callable*[[*Union*[*Type*[*View*], *Callable*[[*View*, *Request*], *Union*[*Coroutine*[*Any*, *Any*, *Response*], *Awaitable*[*Response*]]], *Callable*[[*View*, *Request*, *Any*, *Any*], *Union*[*Coroutine*[*Any*, *Any*, *Response*], *Awaitable*[*Response*]]]]], *Type*[*View*]]

table_route(*table: faust.types.tables.CollectionT*, *shard_param: str = None*, ***, *query_param: str = None*, *match_info: str = None*, *exact_key: str = None*) → Callable[Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]]

Decorate view method to route request to table key destination.

Return type Callable[[Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]], Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]]

command(**options: Any*, *base: Optional[Type[faust.app.base._AppCommand]] = None*, ***kwargs: Any*) → Callable[Callable, Type[faust.app.base._AppCommand]]

Decorate `async def` function to be used as CLI command.

Return type Callable[[Callable], Type[_AppCommand]]

create_event(*key: Union[bytes, faust.types.core._ModelT, Any, None]*, *value: Union[bytes, faust.types.core._ModelT, Any]*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]*, *message: faust.types.tuples.Message*) → faust.types.events.EventT

Create new *faust.Event* object.

Return type *EventT*[]

async start_client() → None

Start the app in Client-Only mode necessary for RPC requests.

Notes

Once started as a client the app cannot be restarted as Server.

Return type None

async maybe_start_client() → None

Start the app in Client-Only mode if not started as Server.

Return type None

trace(*name: str*, *trace_enabled: bool = True*, ***extra_context: Any*) → ContextManager

Return new trace context to trace operation using OpenTracing.

Return type ContextManager[+T_co]

traced(*fun: Callable*, *name: str = None*, *sample_rate: float = 1.0*, ***context: Any*) → Callable

Decorate function to be traced using the OpenTracing API.

Return type Callable

async send(*channel: Union[faust.types.channels.ChannelT, str], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.serializers.SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*) → Awaitable[faust.types.tuples.RecordMetadata]

Send event to channel/topic.

Parameters

- **channel** (Union[*ChannelT*[], str]) – Channel/topic or the name of a topic to send event to.
- **key** (Union[bytes, _ModelT, Any, None]) – Message key.
- **value** (Union[bytes, _ModelT, Any, None]) – Message value.
- **partition** (Optional[int]) – Specific partition to send to. If not set the partition will be chosen by the partitioner.
- **timestamp** (Optional[float]) – Epoch seconds (from Jan 1 1970 UTC) to use as the message timestamp. Defaults to current time.
- **headers** (Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]) – Mapping of key/value pairs, or iterable of key value pairs to use as headers for the message.
- **schema** (Optional[*SchemaT*[~KT, ~VT]]) – *Schema* to use for serialization.
- **key_serializer** (Union[*CodecT*, str, None]) – Serializer to use (if value is not model). Overrides schema if one is specified.
- **value_serializer** (Union[*CodecT*, str, None]) – Serializer to use (if value is not model). Overrides schema if one is specified.
- **callback** (Optional[Callable[[*FutureMessage*[]], Union[None, Awaitable[None]]]]) – Called after the message is fully delivered to the channel, but not to the consumer. Signature must be unary as the *FutureMessage* future is passed to it.

  The resulting *faust.types.tuples.RecordMetadata* object is then available as `fut.result()`.

Return type Awaitable[*RecordMetadata*]

in_transaction

Return `True` if stream is using transactions.

LiveCheck(***kwargs: Any*) → faust.app.base._LiveCheck

Return new LiveCheck instance testing features for this app.

Return type _LiveCheck

maybe_start_producer

Ensure producer is started. :rtype: *ProducerT*[]

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]]*) → bool

Commit offset for acked messages in specified topics'.

> **Warning:** This will commit acked messages in **all topics** if the topics argument is passed in as `None`.

Return type bool

async on_stop() → None
Call when application stops.

Tip: Remember to call `super` if you override this method.

Return type None

on_rebalance_start() → None
Call when rebalancing starts.

Return type None

on_rebalance_return() → None

Return type None

on_rebalance_end() → None
Call when rebalancing is done.

Return type None

FlowControlQueue(*maxsize: int = None*, ***, *clear_on_resume: bool = False*, *loop: asyncio.events.AbstractEventLoop = None*) → mode.utils.queues.ThrowableQueue
Like `asyncio.Queue`, but can be suspended/resumed.

Return type ThrowableQueue

Worker(***kwargs: Any*) → faust.app.base._Worker
Return application worker instance.

Return type _Worker

on_webserver_init(*web: faust.types.web.Web*) → None
Call when the Web server is initializing.

Return type None

property conf
Application configuration. :rtype: *Settings*

property producer
Message producer. :rtype: *ProducerT*[]

property consumer
Message consumer. :rtype: *ConsumerT*[]

property transport
Consumer message transport. :rtype: *TransportT*

logger = <Logger faust.app.base (WARNING)>

property producer_transport
Producer message transport. :rtype: *TransportT*

property cache
Cache backend. :rtype: *CacheBackendT*[]

tables
Map of available tables, and the table manager service.

topics
Topic Conductor.

This is the mediator that moves messages fetched by the Consumer into the streams.

It's also a set of registered topics by string topic name, so you can check if a topic is being consumed from by doing `topic in app.topics`.

property monitor
Monitor keeps stats about what's going on inside the worker. :rtype: *Monitor*[]

flow_control
Flow control of streams.

This object controls flow into stream queues, and can also clear all buffers.

property http_client
HTTP client Session. :rtype: `ClientSession`

assignor
Partition Assignor.

Responsible for partition assignment.

router
Find the node partitioned data belongs to.

The router helps us route web requests to the wanted Faust node. If a topic is sharded by account_id, the router can send us to the Faust worker responsible for any account. Used by the `@app.table_route` decorator.

web
Web driver.

serializers
Return serializer registry.

property label
Return human readable description of application. :rtype: `str`

property shortlabel
Return short description of application. :rtype: `str`

faust.app.router

Route messages to Faust nodes by partitioning.

class faust.app.router.**Router**(*app: faust.types.app.AppT*) → None
Router for `app.router`.

key_store(*table_name: str, key: Union[bytes, faust.types.core._ModelT, Any, None]*) → yarl.URL
Return the URL of web server that hosts key in table.

Return type `URL`

table_metadata(*table_name: str*) → MutableMapping[str, MutableMapping[str, List[int]]]
Return metadata stored for table in the partition assignor.

Return type `MutableMapping[str, MutableMapping[str, List[int]]]`

tables_metadata() → MutableMapping[str, MutableMapping[str, List[int]]]
Return metadata stored for all tables in the partition assignor.

Return type `MutableMapping[str, MutableMapping[str, List[int]]]`

async route_req(*table_name: str, key: Union[bytes, faust.types.core._ModelT, Any, None], web: faust.types.web.Web, request: faust.types.web.Request*) → faust.types.web.Response
Route request to worker having key in table.

Parameters

- **table_name** (str) – Name of the table.
- **key** (Union[bytes, _ModelT, Any, None]) – The key that we want.
- **web** (*Web*) – The currently sued web driver,
- **request** (*Request*) – The web request currently being served.

Return type *Response*

1.6.3 Agents

faust.agents

Agents.

class faust.agents.**Agent**(*fun: Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], *, app: faust.types.app.AppT, name: str = None, channel: Union[str, faust.types.channels.ChannelT] = None, concurrency: int = 1, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, on_error: Callable[[AgentT, BaseException], Awaitable] = None, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, help: str = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, isolated_partitions: bool = False, use_reply_headers: bool = None, **kwargs: Any*) → None
Agent.

This is the type of object returned by the `@app.agent` decorator.

supervisor = None

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of services dependencies required to start agent.

Return type Iterable[ServiceT[]]

async on_start() → None
Call when an agent starts.

Return type None

async on_stop() → None
Call when an agent stops.

Return type None

cancel() → None
Cancel agent and its actor instances running in this process.

Return type None

async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when partitions are revoked.

Return type None

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when partitions are assigned.

Return type None

async on_isolated_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when isolated partitions are revoked.

Return type None

async on_isolated_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when isolated partitions are assigned.

Return type None

async on_shared_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when non-isolated partitions are revoked.

Return type None

async on_shared_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when non-isolated partitions are assigned.

Return type None

info() → Mapping
Return agent attributes as a dictionary.

Return type Mapping[~KT, +VT_co]

clone(*, *cls: Type[faust.types.agents.AgentT] = None*, ***kwargs: Any*) → faust.types.agents.AgentT
Create clone of this agent object.

Keyword arguments can be passed to override any argument supported by *Agent.__init__*.

Return type *AgentT*[]

test_context(*channel: faust.types.channels.ChannelT = None*, *supervisor_strategy: mode.types.supervisors.SupervisorStrategyT = None*, *on_error: Callable[[AgentT, BaseException], Awaitable] = None*, ***kwargs: Any*) → faust.types.agents.AgentTestWrapperT
Create new unit-testing wrapper for this agent.

Return type *AgentTestWrapperT*[]

actor_from_stream(*stream: Optional[faust.types.streams.StreamT]*, ***, *index: int = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, *channel: faust.types.channels.ChannelT = None*) → faust.types.agents.ActorT[Union[AsyncIterable, Awaitable]]
Create new actor from stream.

Return type *ActorT*[]

add_sink(*sink: Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]*) → None
Add new sink to further handle results from this agent.

Return type None

stream(*channel: faust.types.channels.ChannelT = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, ***kwargs: Any*) → faust.types.streams.StreamT
Create underlying stream used by this agent.

Return type *StreamT*[+T_co]

async cast(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*) → None

RPC operation: like `ask()` but do not expect reply.

Cast here is like "casting a spell", and will not expect a reply back from the agent.

Return type None

async ask(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None*) → Any

RPC operation: ask agent for result of processing value.

This version will wait until the result is available and return the processed value.

Return type Any

async ask_nowait(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None, force: bool = False*) → faust.agents.replies.ReplyPromise

RPC operation: ask agent for result of processing value.

This version does not wait for the result to arrive, but instead returns a promise of future evaluation.

Return type *ReplyPromise*

async send(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Send message to topic used by agent.

Return type Awaitable[*RecordMetadata*]

map(*values: Union[AsyncIterable, Iterable], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator

RPC map operation on a list of values.

A map operation iterates over results as they arrive. See `join()` and `kvjoin()` if you want them in order.

Return type AsyncIterator[+T_co]

kvmap(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator[str]

RPC map operation on a list of `(key, value)` pairs.

A map operation iterates over results as they arrive. See `join()` and `kvjoin()` if you want them in order.

Return type AsyncIterator[str]

async join(*values: Union[AsyncIterable[Union[bytes, faust.types.core._ModelT, Any]], Iterable[Union[bytes, faust.types.core._ModelT, Any]]], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]

RPC map operation on a list of values.

A join returns the results in order, and only returns once all values have been processed.

Return type List[Any]

async kvjoin(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]

RPC map operation on list of `(key, value)` pairs.

A join returns the results in order, and only returns once all values have been processed.

Return type List[Any]

get_topic_names() → Iterable[str]

Return list of topic names this agent subscribes to.

Return type Iterable[str]

property channel

Return channel used by agent. :rtype: *ChannelT*[]

property channel_iterator

Return channel agent iterates over. :rtype: AsyncIterator[+T_co]

property label

Return human-readable description of agent. :rtype: str

property shortlabel

Return short description of agent. :rtype: str

logger = <Logger faust.agents.agent (WARNING)>

faust.agents.**AgentFun**

alias of typing.Callable

class faust.agents.**AgentT**(*fun: Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], *, name: str = None, app: faust.types.agents._AppT = None, channel: Union[str, faust.types.channels.ChannelT] = None, concurrency: int = 1, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, on_error: Callable[[AgentT, BaseException], Awaitable] = None, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, help: str = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, isolated_partitions: bool = False, **kwargs: Any*) → None

abstract test_context(*channel: faust.types.channels.ChannelT = None, supervisor_strategy: mode.types.supervisors.SupervisorStrategyT = None, **kwargs: Any*) → faust.types.agents.AgentTestWrapperT

Return type *AgentTestWrapperT*[]

abstract add_sink(*sink: Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]*) → None

Return type None

abstract stream(***kwargs: Any*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async cast(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*) → None

Return type None

abstract async ask(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None*) → Any

Return type Any

abstract async send(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract async map(*values: Union[AsyncIterable, Iterable], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator

abstract async kvmap(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator[str]

abstract async join(*values: Union[AsyncIterable[Union[bytes, faust.types.core._ModelT, Any]], Iterable[Union[bytes, faust.types.core._ModelT, Any]]], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]

Return type List[Any]

abstract async kvjoin(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]

Return type List[Any]

abstract info() → Mapping

Return type Mapping[~KT, +VT_co]

abstract clone(*, *cls: Type[AgentT] = None*, ***kwargs: Any*) → faust.types.agents.AgentT

Return type *AgentT*[]

abstract get_topic_names() → Iterable[str]

Return type Iterable[str]

abstract property channel

Return type *ChannelT*[]

abstract property channel_iterator

Return type AsyncIterator[+T_co]

faust.agents.**current_agent**() → Optional[faust.types.agents.AgentT]

Return type Optional[*AgentT*[]]

class faust.agents.**AgentManager**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None

Agent manager.

async on_start() → None

Call when agents are being started.

Return type None

service_reset() → None

Reset service state on restart.

Return type None

async on_stop() → None

Call when agents are being stopped.

Return type None

async stop() → None

Stop all running agents.

Return type None

cancel() → None

Cancel all running agents.

Return type None

update_topic_index() → None

Update indices.

Return type None

async on_rebalance(*revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None

Call when a rebalance is needed.

Return type None

logger = <Logger faust.agents.manager (WARNING)>

class faust.agents.**AgentManagerT**(*, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None

abstract async on_rebalance(*revoked: Set[faust.types.tuples.TP]*, *newly_assigned: Set[faust.types.tuples.TP]*) → None

Return type None

class faust.agents.**ReplyConsumer**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None

Consumer responsible for redelegation of replies received.

logger = <Logger faust.agents.replies (WARNING)>

async on_start() → None

Call when reply consumer starts.

Return type None

async add(*correlation_id: str*, *promise: faust.agents.replies.ReplyPromise*) → None

Register promise to start tracking when it arrives.

Return type None

faust.agents.actor

Actor - Individual Agent instances.

class faust.agents.actor.**Actor**(*agent: faust.types.agents.AgentT*, *stream: faust.types.streams.StreamT*, *it: _T*, *index: int = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, ***kwargs: Any*) → None

An actor is a specific agent instance.

mundane_level = 'debug'

async on_start() → None

Call when actor is starting.

Return type None

async on_stop() → None

Call when actor is being stopped.

Return type None

async on_isolated_partition_revoked(*tp: faust.types.tuples.TP*) → None

Call when an isolated partition is being revoked.

Return type None

async on_isolated_partition_assigned(*tp: faust.types.tuples.TP*) → None

Call when an isolated partition is being assigned.

Return type None

cancel() → None

Tell actor to stop reading from the stream.

Return type None

property label
Return human readable description of actor. :rtype: str

logger = <Logger faust.agents.actor (WARNING)>

class faust.agents.actor.**AsyncIterableActor**(*agent: faust.types.agents.AgentT*, *stream: faust.types.streams.StreamT*, *it: _T*, *index: int = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, ***kwargs: Any*) → None
Used for agent function that yields.

logger = <Logger faust.agents.actor (WARNING)>

class faust.agents.actor.**AwaitableActor**(*agent: faust.types.agents.AgentT*, *stream: faust.types.streams.StreamT*, *it: _T*, *index: int = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, ***kwargs: Any*) → None
Used for actor function that do not yield.

logger = <Logger faust.agents.actor (WARNING)>

faust.agents.agent

Agent implementation.

class faust.agents.agent.**Agent**(*fun: Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], *, app: faust.types.app.AppT, name: str = None, channel: Union[str, faust.types.channels.ChannelT] = None, concurrency: int = 1, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, on_error: Callable[[AgentT, BaseException], Awaitable] = None, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, help: str = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, isolated_partitions: bool = False, use_reply_headers: bool = None, **kwargs: Any*) → None
Agent.

This is the type of object returned by the `@app.agent` decorator.

supervisor = None

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of services dependencies required to start agent.

Return type Iterable[ServiceT[]]

async on_start() → None
Call when an agent starts.

Return type None

async on_stop() → None
Call when an agent stops.

Return type None

cancel() → None
Cancel agent and its actor instances running in this process.

Return type None

async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when partitions are revoked.

Return type None

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when partitions are assigned.

Return type None

async on_isolated_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when isolated partitions are revoked.

Return type None

async on_isolated_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when isolated partitions are assigned.

Return type None

async on_shared_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when non-isolated partitions are revoked.

Return type None

async on_shared_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when non-isolated partitions are assigned.

Return type None

info() → Mapping
Return agent attributes as a dictionary.

Return type Mapping[~KT, +VT_co]

clone(*, *cls: Type[faust.types.agents.AgentT] = None*, ***kwargs: Any*) → faust.types.agents.AgentT
Create clone of this agent object.

Keyword arguments can be passed to override any argument supported by *Agent.__init__*.

Return type *AgentT*[]

test_context(*channel: faust.types.channels.ChannelT = None*, *supervisor_strategy: mode.types.supervisors.SupervisorStrategyT = None*, *on_error: Callable[[AgentT, BaseException], Awaitable] = None*, ***kwargs: Any*) → faust.types.agents.AgentTestWrapperT
Create new unit-testing wrapper for this agent.

Return type *AgentTestWrapperT*[]

actor_from_stream(*stream: Optional[faust.types.streams.StreamT]*, ***, *index: int = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, *channel: faust.types.channels.ChannelT = None*) → faust.types.agents.ActorT[Union[AsyncIterable, Awaitable]]
Create new actor from stream.

Return type *ActorT*[]

add_sink(*sink: Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]*) → None
Add new sink to further handle results from this agent.

Return type None

stream(*channel: faust.types.channels.ChannelT = None, active_partitions: Set[faust.types.tuples.TP] = None, **kwargs: Any*) → faust.types.streams.StreamT
Create underlying stream used by this agent.

Return type *StreamT*[+T_co]

async cast(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*) → None
RPC operation: like *ask()* but do not expect reply.

Cast here is like "casting a spell", and will not expect a reply back from the agent.

Return type None

async ask(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None*) → Any
RPC operation: ask agent for result of processing value.

This version will wait until the result is available and return the processed value.

Return type Any

async ask_nowait(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None, force: bool = False*) → faust.agents.replies.ReplyPromise
RPC operation: ask agent for result of processing value.

This version does not wait for the result to arrive, but instead returns a promise of future evaluation.

Return type *ReplyPromise*

async send(*, *key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]
Send message to topic used by agent.

Return type Awaitable[*RecordMetadata*]

map(*values: Union[AsyncIterable, Iterable], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator
RPC map operation on a list of values.

A map operation iterates over results as they arrive. See *join()* and *kvjoin()* if you want them in order.

Return type AsyncIterator[+T_co]

kvmap(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator[str]
RPC map operation on a list of `(key, value)` pairs.

A map operation iterates over results as they arrive. See *join()* and *kvjoin()* if you want them in order.

Return type AsyncIterator[str]

async join(*values: Union[AsyncIterable[Union[bytes, faust.types.core._ModelT, Any]], Iterable[Union[bytes, faust.types.core._ModelT, Any]]], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]

RPC map operation on a list of values.

A join returns the results in order, and only returns once all values have been processed.

Return type List[Any]

async kvjoin(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]

RPC map operation on list of `(key, value)` pairs.

A join returns the results in order, and only returns once all values have been processed.

Return type List[Any]

get_topic_names() → Iterable[str]

Return list of topic names this agent subscribes to.

Return type Iterable[str]

property channel

Return channel used by agent. :rtype: *ChannelT*[]

property channel_iterator

Return channel agent iterates over. :rtype: AsyncIterator[+T_co]

property label

Return human-readable description of agent. :rtype: str

property shortlabel

Return short description of agent. :rtype: str

logger = <Logger faust.agents.agent (WARNING)>

faust.agents.manager

Agent manager.

class faust.agents.manager.**AgentManager**(*app: faust.types.app.AppT, **kwargs: Any*) → None

Agent manager.

async on_start() → None

Call when agents are being started.

Return type None

service_reset() → None

Reset service state on restart.

Return type None

async on_stop() → None

Call when agents are being stopped.

Return type None

async stop() → None
Stop all running agents.

Return type None

cancel() → None
Cancel all running agents.

Return type None

update_topic_index() → None
Update indices.

Return type None

async on_rebalance(*revoked: Set[faust.types.tuples.TP]*, *newly_assigned: Set[faust.types.tuples.TP]*) → None
Call when a rebalance is needed.

Return type None

logger = <Logger faust.agents.manager (WARNING)>

faust.agents.models

Models used by agents internally.

class faust.agents.models.**ReqRepRequest**(*value*, *reply_to*, *correlation_id*, ***, *__strict__=True*, *__faust=None*, ***kwargs*) → None
Value wrapped in a Request-Reply request.

value
Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.

- **parent** (FieldDescriptorT) – parent field if any.

reply_to

correlation_id

asdict()

class faust.agents.models.**ReqRepResponse**(*value*, *correlation_id*, *key=None*, *, *__strict__=True*, *__faust=None*, ***kwargs*) → None

Request-Reply response.

key

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

value

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

correlation_id

asdict()

faust.agents.replies

Agent replies: waiting for replies, sending them, etc.

class faust.agents.replies.**ReplyPromise**(*reply_to: str*, *correlation_id: str*, ***kwargs: Any*) → None

Reply promise can be await-ed to wait until result ready.

fulfill(*correlation_id: str*, *value: Any*) → None

Fulfill promise: a reply was received.

Return type None

class faust.agents.replies.**BarrierState**(*reply_to: str*, ***kwargs: Any*) → None

State of pending/complete barrier.

A barrier is a synchronization primitive that will wait until a group of coroutines have completed.

size = 0

This is the size while the messages are being sent. (it's a tentative total, added to until the total is finalized).

total = 0

This is the actual total when all messages have been sent. It's set by *finalize()*.

fulfilled = 0

The number of results we have received.

pending = None

Set of pending replies that this barrier is composed of.

add(*p: faust.agents.replies.ReplyPromise*) → None
Add promise to barrier.

Note: You can only add promises before the barrier is finalized using *finalize()*.

Return type None

finalize() → None
Finalize this barrier.

After finalization you can not grow or shrink the size of the barrier.

Return type None

fulfill(*correlation_id: str*, *value: Any*) → None
Fulfill one of the promises in this barrier.

Once all promises in this barrier is fulfilled, the barrier will be ready.

Return type None

get_nowait() → faust.agents.replies.ReplyTuple
Return next reply, or raise asyncio.QueueEmpty.

Return type ReplyTuple

iterate() → AsyncIterator[faust.agents.replies.ReplyTuple]
Iterate over results as they arrive.

Return type AsyncIterator[ReplyTuple]

class faust.agents.replies.**ReplyConsumer**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Consumer responsible for redelegation of replies received.

logger = <Logger faust.agents.replies (WARNING)>

async on_start() → None
Call when reply consumer starts.

Return type None

async add(*correlation_id: str*, *promise: faust.agents.replies.ReplyPromise*) → None
Register promise to start tracking when it arrives.

Return type None

1.6.4 Fixups

faust.fixups

Transport registry.

faust.fixups.**by_name**(*name: Union[_T, str]*) → _T

Return type ~_T

faust.fixups.**by_url**(*url: Union[str, yarl.URL]*) → _T
Get class associated with URL (scheme is used as alias key).

Return type ~_T

faust.fixups.**fixups**(*app: faust.types.app.AppT*) → Iterator[faust.types.fixups.FixupT]
Iterate over enabled fixups.

Fixups are installed by setuptools, using the 'faust.fixups' namespace.

Fixups modify the Faust library to work with frameworks such as Django.

Return type Iterator[*FixupT*]

faust.fixups.base

Fixups - Base implementation.

class faust.fixups.base.**Fixup**(*app: faust.types.app.AppT*) → None
Base class for fixups.

Fixups are things that hook into Faust to make things work for other frameworks, such as Django.

enabled() → bool
Return if fixup should be enabled in this environment.

Return type bool

autodiscover_modules() → Iterable[str]
Return list of additional autodiscover modules.

Return type Iterable[str]

on_worker_init() → None
Call when initializing worker/CLI commands.

Return type None

faust.fixups.django

Django Fixups - Integration with Django.

class faust.fixups.django.**Fixup**(*app: faust.types.app.AppT*) → None
Django fixup.

This fixup is enabled if

1) the DJANGO_SETTINGS_MODULE environment variable is set,

2) the django package is installed.

Once enabled it will modify the following features:

- Autodiscovery

  If faust.App(autodiscovery=True), the Django fixup will automatically autodiscover agents/tasks/web views, and so on found in installed Django apps.

- Setup

  The Django machinery will be set up when Faust commands are executed.

enabled() → bool
Return True if Django is used in this environment.

Return type bool

wait_for_django() → None

Return type None

autodiscover_modules() → Iterable[str]
Return list of additional autodiscover modules.

For Django we run autodiscovery in all packages listed in the INSTALLED_APPS setting (with support for custom app configurations).

Return type Iterable[str]

on_worker_init() → None
Initialize Django before worker/CLI command starts.

Return type None

apps
Return the Django app registry.

settings
Return the Django settings object.

1.6.5 LiveCheck

faust.livecheck

LiveCheck - End-to-end testing of asynchronous systems.

class faust.livecheck.**LiveCheck**(*id: str*, ***, *test_topic_name: str = None*, *bus_topic_name: str = None*, *report_topic_name: str = None*, *bus_concurrency: int = None*, *test_concurrency: int = None*, *send_reports: bool = None*, ***kwargs: Any*) → None
LiveCheck application.

SCAN_CATEGORIES = ['faust.agent', 'faust.command', 'faust.page', 'faust.service', 'fau

class Signal(*name: str = ''*, *case: faust.livecheck.signals._Case = None*, *index: int = -1*) → None
Signal for test case using Kafka.

Used to wait for something to happen elsewhere.

async send(*value: VT = None*, ***, *key: Any = None*, *force: bool = False*) → None
Notify test that this signal is now complete.
Return type None

async wait(***, *key: Any = None*, *timeout: Union[datetime.timedelta, float, str] = None*) → VT
Wait for signal to be completed.
Return type ~VT

class Case(***, *app: faust.livecheck.case._LiveCheck*, *name: str*, *probability: float = None*, *warn_stalled_after: Union[datetime.timedelta, float, str] = None*, *active: bool = None*, *signals: Iterable[faust.livecheck.signals.BaseSignal] = None*, *test_expires: Union[datetime.timedelta, float, str] = None*, *frequency: Union[datetime.timedelta, float, str] = None*, *realtime_logs: bool = None*, *max_history: int = None*, *max_consecutive_failures: int = None*, *url_timeout_total: float = None*, *url_timeout_connect: float = None*, *url_error_retries: int = None*, *url_error_delay_min: float = None*, *url_error_delay_backoff: float = None*, *url_error_delay_max: float = None*, ***kwargs: Any*) → None
LiveCheck test case.

Runner
alias of *faust.livecheck.runners.TestRunner*

active = True

consecutive_failures = 0

property current_execution
Return the currently executing *TestRunner* in this task. :rtype: Optional[*TestRunner*]

property current_test
Return the currently active test in this task (if any). :rtype: Optional[*TestExecution*]

async execute(*test: faust.livecheck.models.TestExecution*) → None
Execute test using *TestRunner*.
Return type None

frequency = None

frequency_avg = None

async get_url(*url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform GET request using HTTP client.
Return type Optional[bytes]

property label
Return human-readable label for this test case. :rtype: str

last_fail = None

last_test_received = None

latency_avg = None

logger = <Logger faust.livecheck.case (WARNING)>

max_consecutive_failures = 30

max_history = 100

maybe_trigger(*id: str = None, *args: Any, **kwargs: Any*) → AsyncGenerator[Optional[faust.livecheck.models.TestExecution], None]
Schedule test execution, or not, based on probability setting.
Return type AsyncGenerator[Optional[*TestExecution*], None]

async on_suite_fail(*exc: faust.livecheck.exceptions.SuiteFailed, new_state: faust.livecheck.models.State = <State.FAIL: 'FAIL'>*) → None
Call when the suite fails.
Return type None

async on_test_error(*runner: faust.livecheck.runners.TestRunner, exc: BaseException*) → None
Call when a test execution raises an exception.
Return type None

async on_test_failed(*runner: faust.livecheck.runners.TestRunner, exc: BaseException*) → None
Call when invariant in test execution fails.
Return type None

async on_test_pass(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test execution passes.
Return type None

async on_test_skipped(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test is skipped.
Return type None

async on_test_start(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test starts executing.
Return type None

async on_test_timeout(*runner: faust.livecheck.runners.TestRunner*, *exc: BaseException*) → None
Call when a test execution times out.
Return type None

async post_report(*report: faust.livecheck.models.TestReport*) → None
Publish test report.
Return type None

async post_url(*url: Union[str, yarl.URL]*, ***kwargs: Any*) → Optional[bytes]
Perform POST request using HTTP client.
Return type Optional[bytes]

probability = 0.5

realtime_logs = False

async resolve_signal(*key: str*, *event: faust.livecheck.models.SignalEvent*) → None
Mark test execution signal as resolved.
Return type None

async run(**test_args: Any*, ***test_kwargs: Any*) → None
Override this to define your test case.
Return type None

runtime_avg = None

property seconds_since_last_fail
Return number of seconds since any test failed. :rtype: Optional[float]

state = 'INIT'

state_transition_delay = 60.0

test_expires = datetime.timedelta(0, 10800)

total_failures = 0

async trigger(*id: str = None*, **args: Any*, ***kwargs: Any*) → faust.livecheck.models.TestExecution
Schedule test execution ASAP.
Return type *TestExecution*

url_error_delay_backoff = 1.5

url_error_delay_max = 5.0

url_error_delay_min = 0.5

url_error_retries = 10

async url_request(*method: str*, *url: Union[str, yarl.URL]*, ***kwargs: Any*) → Optional[bytes]
Perform URL request using HTTP client.
Return type Optional[bytes]

url_timeout_connect = None

url_timeout_total = 300.0

warn_stalled_after = 1800.0

classmethod for_app(*app: faust.types.app.AppT*, ***, *prefix: str = 'livecheck-'*, *web_port: int = 9999*, *test_topic_name: str = None*, *bus_topic_name: str = None*, *report_topic_name: str = None*, *bus_concurrency: int = None*, *test_concurrency: int = None*, *send_reports: bool = None*, ***kwargs: Any*) → faust.livecheck.app.LiveCheck

Create LiveCheck application targeting specific app.

The target app will be used to configure the LiveCheck app.

Return type *LiveCheck*[]

test_topic_name = 'livecheck'

bus_topic_name = 'livecheck-bus'

report_topic_name = 'livecheck-report'

bus_concurrency = 30

Number of concurrent actors processing signal events.

test_concurrency = 100

Number of concurrent actors executing test cases.

send_reports = True

Unset this if you don't want reports to be sent to the *report_topic_name* topic.

property current_test

Return the current test context (if any). :rtype: Optional[*TestExecution*]

on_produce_attach_test_headers(*sender: faust.types.app.AppT*, *key: bytes = None*, *value: bytes = None*, *partition: int = None*, *timestamp: float = None*, *headers: List[Tuple[str, bytes]] = None*, ***kwargs: Any*) → None

Attach test headers to Kafka produce requests.

Return type None

case(***, *name: str = None*, *probability: float = None*, *warn_stalled_after: Union[datetime.timedelta, float, str] = datetime.timedelta(0, 1800)*, *active: bool = None*, *test_expires: Union[datetime.timedelta, float, str] = None*, *frequency: Union[datetime.timedelta, float, str] = None*, *max_history: int = None*, *max_consecutive_failures: int = None*, *url_timeout_total: float = None*, *url_timeout_connect: float = None*, *url_error_retries: float = None*, *url_error_delay_min: float = None*, *url_error_delay_backoff: float = None*, *url_error_delay_max: float = None*, *base: Type[faust.livecheck.case.Case] = faust.livecheck.case.Case*) → Callable[Type, faust.livecheck.case.Case]

Decorate class to be used as a test case.

Return type Callable[[Type[+CT_co]], *Case*[]]

Returns *faust.livecheck.Case*.

add_case(*case: faust.livecheck.case.Case*) → faust.livecheck.case.Case

Add and register new test case.

Return type *Case*[]

async post_report(*report: faust.livecheck.models.TestReport*) → None

Publish test report to reporting topic.

Return type None

logger = <Logger faust.livecheck.app (WARNING)>

async on_start() → None

Call when LiveCheck application starts.

Return type None

async on_started() → None
Call when LiveCheck application is fully started.

Return type None

bus
Topic used for signal communication.

pending_tests
Topic used to keep pending test executions.

reports
Topic used to log test reports.

class faust.livecheck.**Case**(*, *app: faust.livecheck.case._LiveCheck*, *name: str*, *probability: float = None*, *warn_stalled_after: Union[datetime.timedelta, float, str] = None*, *active: bool = None*, *signals: Iterable[faust.livecheck.signals.BaseSignal] = None*, *test_expires: Union[datetime.timedelta, float, str] = None*, *frequency: Union[datetime.timedelta, float, str] = None*, *realtime_logs: bool = None*, *max_history: int = None*, *max_consecutive_failures: int = None*, *url_timeout_total: float = None*, *url_timeout_connect: float = None*, *url_error_retries: int = None*, *url_error_delay_min: float = None*, *url_error_delay_backoff: float = None*, *url_error_delay_max: float = None*, ***kwargs: Any*) → None
LiveCheck test case.

Runner
alias of *faust.livecheck.runners.TestRunner*

state = 'INIT'
Current state of this test.

last_test_received = None
The warn_stalled_after timer uses this to keep track of either when a test was last received, or the last time the timer timed out.

last_fail = None
Timestamp of when the suite last failed.

runtime_avg = None

latency_avg = None

frequency_avg = None

state_transition_delay = 60.0

consecutive_failures = 0

total_failures = 0

name = None
Name of the test If not set this will be generated out of the subclass name.

active = True

probability = 0.5
Probability of test running when live traffic is going through.

warn_stalled_after = 1800.0
Timeout in seconds for when after we warn that nothing is processing.

test_expires = datetime.timedelta(0, 10800)

frequency = None
How often we execute the test using fake data (define Case.make_fake_request()).

Set to None if production traffic is frequent enough to satisfy *warn_stalled_after*.

realtime_logs = False

max_history = 100
Max items to store in `latency_history` and `runtime_history`.

max_consecutive_failures = 30

url_timeout_total = 300.0

url_timeout_connect = None

url_error_retries = 10

url_error_delay_min = 0.5

url_error_delay_backoff = 1.5

url_error_delay_max = 5.0

maybe_trigger(*id: str = None*, **args: Any*, ***kwargs: Any*) → AsyncGenerator[Optional[faust.livecheck.models.TestExecution], None]
Schedule test execution, or not, based on probability setting.

Return type AsyncGenerator[Optional[*TestExecution*], None]

async trigger(*id: str = None*, **args: Any*, ***kwargs: Any*) → faust.livecheck.models.TestExecution
Schedule test execution ASAP.

Return type *TestExecution*

logger = <Logger faust.livecheck.case (WARNING)>

async run(**test_args: Any*, ***test_kwargs: Any*) → None
Override this to define your test case.

Return type None

async resolve_signal(*key: str*, *event: faust.livecheck.models.SignalEvent*) → None
Mark test execution signal as resolved.

Return type None

async execute(*test: faust.livecheck.models.TestExecution*) → None
Execute test using *TestRunner*.

Return type None

async on_test_start(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test starts executing.

Return type None

async on_test_skipped(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test is skipped.

Return type None

async on_test_failed(*runner: faust.livecheck.runners.TestRunner*, *exc: BaseException*) → None
Call when invariant in test execution fails.

Return type None

async on_test_error(*runner: faust.livecheck.runners.TestRunner*, *exc: BaseException*) → None
Call when a test execution raises an exception.

Return type None

async on_test_timeout(*runner: faust.livecheck.runners.TestRunner*, *exc: BaseException*) → None
Call when a test execution times out.

Return type None

async on_test_pass(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test execution passes.

Return type None

async post_report(*report: faust.livecheck.models.TestReport*) → None
Publish test report.

Return type None

async on_suite_fail(*exc: faust.livecheck.exceptions.SuiteFailed*, *new_state: faust.livecheck.models.State = <State.FAIL: 'FAIL'>*) → None
Call when the suite fails.

Return type None

property seconds_since_last_fail
Return number of seconds since any test failed. :rtype: Optional[float]

async get_url(*url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform GET request using HTTP client.

Return type Optional[bytes]

async post_url(*url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform POST request using HTTP client.

Return type Optional[bytes]

async url_request(*method: str, url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform URL request using HTTP client.

Return type Optional[bytes]

property current_test
Return the currently active test in this task (if any). :rtype: Optional[*TestExecution*]

property current_execution
Return the currently executing *TestRunner* in this task. :rtype: Optional[*TestRunner*]

property label
Return human-readable label for this test case. :rtype: str

faust.livecheck.**current_test**() → Optional[faust.livecheck.models.TestExecution]
Return information about the current test (if any).

Return type Optional[*TestExecution*]

class faust.livecheck.**TestRunner**(*case: faust.livecheck.runners._Case*, *test: faust.livecheck.models.TestExecution*, *started: float*) → None
Execute and keep track of test instance.

state = 'INIT'

report = None

error = None

async execute() → None
Execute this test.

Return type None

async skip(*reason: str*) → NoReturn
Skip this test execution.

Return type _NoReturn

async on_skipped(*exc: faust.livecheck.exceptions.TestSkipped*) → None
Call when a test execution was skipped.

Return type None

async on_start() → None
Call when a test starts executing.

Return type None

async on_signal_wait(*signal: faust.livecheck.signals.BaseSignal*, *timeout: float*) → None
Call when the test is waiting for a signal.

Return type None

async on_signal_received(*signal: faust.livecheck.signals.BaseSignal*, *time_start: float*, *time_end: float*) → None
Call when a signal related to this test is received.

Return type None

async on_failed(*exc: BaseException*) → None
Call when an invariant in the test has failed.

Return type None

async on_error(*exc: BaseException*) → None
Call when test execution raises error.

Return type None

async on_timeout(*exc: BaseException*) → None
Call when test execution times out.

Return type None

async on_pass() → None
Call when test execution returns successfully.

Return type None

log_info(*msg: str*, **args: Any*) → None
Log information related to the current execution.

Return type None

end() → None
End test execution.

Return type None

class faust.livecheck.**Signal**(*name: str = ''*, *case: faust.livecheck.signals._Case = None*, *index: int = -1*) → None
Signal for test case using Kafka.

Used to wait for something to happen elsewhere.

async send(*value: VT = None*, ***, *key: Any = None*, *force: bool = False*) → None
Notify test that this signal is now complete.

Return type None

async wait(***, *key: Any = None*, *timeout: Union[datetime.timedelta, float, str] = None*) → VT
Wait for signal to be completed.

Return type ~VT

faust.livecheck.app

LiveCheck - Faust Application.

class faust.livecheck.app.**LiveCheck**(*id: str*, ***, *test_topic_name: str = None*, *bus_topic_name: str = None*, *report_topic_name: str = None*, *bus_concurrency: int = None*, *test_concurrency: int = None*, *send_reports: bool = None*, ***kwargs: Any*) → None
LiveCheck application.

SCAN_CATEGORIES = ['faust.agent', 'faust.command', 'faust.page', 'faust.service', 'fau

class Signal(*name: str = ''*, *case: faust.livecheck.signals._Case = None*, *index: int = -1*) → None
Signal for test case using Kafka.

Used to wait for something to happen elsewhere.

async send(*value: VT = None*, ***, *key: Any = None*, *force: bool = False*) → None
Notify test that this signal is now complete.

Return type None

async wait(***, *key: Any = None*, *timeout: Union[datetime.timedelta, float, str] = None*) → VT
Wait for signal to be completed.

Return type ~VT

class Case(***, *app: faust.livecheck.case._LiveCheck*, *name: str*, *probability: float = None*, *warn_stalled_after: Union[datetime.timedelta, float, str] = None*, *active: bool = None*, *signals: Iterable[faust.livecheck.signals.BaseSignal] = None*, *test_expires: Union[datetime.timedelta, float, str] = None*, *frequency: Union[datetime.timedelta, float, str] = None*, *realtime_logs: bool = None*, *max_history: int = None*, *max_consecutive_failures: int = None*, *url_timeout_total: float = None*, *url_timeout_connect: float = None*, *url_error_retries: int = None*, *url_error_delay_min: float = None*, *url_error_delay_backoff: float = None*, *url_error_delay_max: float = None*, ***kwargs: Any*) → None
LiveCheck test case.

Runner
alias of *faust.livecheck.runners.TestRunner*

active = True

consecutive_failures = 0

property current_execution
Return the currently executing TestRunner in this task. :rtype: Optional[*TestRunner*]

property current_test
Return the currently active test in this task (if any). :rtype: Optional[*TestExecution*]

async execute(*test: faust.livecheck.models.TestExecution*) → None
Execute test using TestRunner.

Return type None

frequency = None

frequency_avg = None

async get_url(*url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform GET request using HTTP client.
Return type Optional[bytes]

property label
Return human-readable label for this test case. :rtype: str

last_fail = None

last_test_received = None

latency_avg = None

logger = <Logger faust.livecheck.case (WARNING)>

max_consecutive_failures = 30

max_history = 100

maybe_trigger(*id: str = None, *args: Any, **kwargs: Any*) → AsyncGenerator[Optional[faust.livecheck.models.TestExecution], None]
Schedule test execution, or not, based on probability setting.
Return type AsyncGenerator[Optional[*TestExecution*], None]

async on_suite_fail(*exc: faust.livecheck.exceptions.SuiteFailed, new_state: faust.livecheck.models.State = <State.FAIL: 'FAIL'>*) → None
Call when the suite fails.
Return type None

async on_test_error(*runner: faust.livecheck.runners.TestRunner, exc: BaseException*) → None
Call when a test execution raises an exception.
Return type None

async on_test_failed(*runner: faust.livecheck.runners.TestRunner, exc: BaseException*) → None
Call when invariant in test execution fails.
Return type None

async on_test_pass(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test execution passes.
Return type None

async on_test_skipped(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test is skipped.
Return type None

async on_test_start(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test starts executing.
Return type None

async on_test_timeout(*runner: faust.livecheck.runners.TestRunner, exc: BaseException*) → None
Call when a test execution times out.
Return type None

async post_report(*report: faust.livecheck.models.TestReport*) → None
Publish test report.
Return type None

async post_url(*url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform POST request using HTTP client.
Return type Optional[bytes]

probability = 0.5

realtime_logs = False

async resolve_signal(*key: str*, *event: faust.livecheck.models.SignalEvent*) → None
Mark test execution signal as resolved.
Return type None

async run(**test_args: Any*, ***test_kwargs: Any*) → None
Override this to define your test case.
Return type None

runtime_avg = None

property seconds_since_last_fail
Return number of seconds since any test failed. :rtype: Optional[float]

state = 'INIT'

state_transition_delay = 60.0

test_expires = datetime.timedelta(0, 10800)

total_failures = 0

async trigger(*id: str = None*, **args: Any*, ***kwargs: Any*) → faust.livecheck.models.TestExecution
Schedule test execution ASAP.
Return type *TestExecution*

url_error_delay_backoff = 1.5

url_error_delay_max = 5.0

url_error_delay_min = 0.5

url_error_retries = 10

async url_request(*method: str, url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform URL request using HTTP client.
Return type Optional[bytes]

url_timeout_connect = None

url_timeout_total = 300.0

warn_stalled_after = 1800.0

classmethod for_app(*app: faust.types.app.AppT*, ***, *prefix: str = 'livecheck-'*, *web_port: int = 9999*, *test_topic_name: str = None*, *bus_topic_name: str = None*, *report_topic_name: str = None*, *bus_concurrency: int = None*, *test_concurrency: int = None*, *send_reports: bool = None*, ***kwargs: Any*) → faust.livecheck.app.LiveCheck
Create LiveCheck application targeting specific app.

The target app will be used to configure the LiveCheck app.

Return type *LiveCheck*[]

test_topic_name = 'livecheck'

bus_topic_name = 'livecheck-bus'

report_topic_name = 'livecheck-report'

bus_concurrency = 30
Number of concurrent actors processing signal events.

test_concurrency = 100
Number of concurrent actors executing test cases.

send_reports = True
Unset this if you don't want reports to be sent to the *report_topic_name* topic.

property current_test
Return the current test context (if any). :rtype: Optional[*TestExecution*]

on_produce_attach_test_headers(*sender: faust.types.app.AppT*, *key: bytes = None*, *value: bytes = None*, *partition: int = None*, *timestamp: float = None*, *headers: List[Tuple[str, bytes]] = None*, ***kwargs: Any*) → None
Attach test headers to Kafka produce requests.

Return type None

case(***, *name: str = None*, *probability: float = None*, *warn_stalled_after: Union[datetime.timedelta, float, str] = datetime.timedelta(0, 1800)*, *active: bool = None*, *test_expires: Union[datetime.timedelta, float, str] = None*, *frequency: Union[datetime.timedelta, float, str] = None*, *max_history: int = None*, *max_consecutive_failures: int = None*, *url_timeout_total: float = None*, *url_timeout_connect: float = None*, *url_error_retries: float = None*, *url_error_delay_min: float = None*, *url_error_delay_backoff: float = None*, *url_error_delay_max: float = None*, *base: Type[faust.livecheck.case.Case] = faust.livecheck.case.Case*) → Callable[Type, faust.livecheck.case.Case]
Decorate class to be used as a test case.

Return type Callable[[Type[+CT_co]], *Case*[]]

Returns *faust.livecheck.Case*.

add_case(*case: faust.livecheck.case.Case*) → faust.livecheck.case.Case
Add and register new test case.

Return type *Case*[]

async post_report(*report: faust.livecheck.models.TestReport*) → None
Publish test report to reporting topic.

Return type None

logger = <Logger faust.livecheck.app (WARNING)>

async on_start() → None
Call when LiveCheck application starts.

Return type None

async on_started() → None
Call when LiveCheck application is fully started.

Return type None

bus
Topic used for signal communication.

pending_tests
Topic used to keep pending test executions.

reports
Topic used to log test reports.

faust.livecheck.case

LiveCheck - Test cases.

class faust.livecheck.case.**Case**(*, *app: faust.livecheck.case._LiveCheck*, *name: str*, *probability: float = None*, *warn_stalled_after: Union[datetime.timedelta, float, str] = None*, *active: bool = None*, *signals: Iterable[faust.livecheck.signals.BaseSignal] = None*, *test_expires: Union[datetime.timedelta, float, str] = None*, *frequency: Union[datetime.timedelta, float, str] = None*, *realtime_logs: bool = None*, *max_history: int = None*, *max_consecutive_failures: int = None*, *url_timeout_total: float = None*, *url_timeout_connect: float = None*, *url_error_retries: int = None*, *url_error_delay_min: float = None*, *url_error_delay_backoff: float = None*, *url_error_delay_max: float = None*, ***kwargs: Any*) → None

LiveCheck test case.

Runner
alias of *faust.livecheck.runners.TestRunner*

state = 'INIT'
Current state of this test.

last_test_received = None
The warn_stalled_after timer uses this to keep track of either when a test was last received, or the last time the timer timed out.

last_fail = None
Timestamp of when the suite last failed.

runtime_avg = None

latency_avg = None

frequency_avg = None

state_transition_delay = 60.0

consecutive_failures = 0

total_failures = 0

name = None
Name of the test If not set this will be generated out of the subclass name.

active = True

probability = 0.5
Probability of test running when live traffic is going through.

warn_stalled_after = 1800.0
Timeout in seconds for when after we warn that nothing is processing.

test_expires = datetime.timedelta(0, 10800)

frequency = None
How often we execute the test using fake data (define Case.make_fake_request()).

Set to None if production traffic is frequent enough to satisfy *warn_stalled_after*.

realtime_logs = False

max_history = 100
Max items to store in `latency_history` and `runtime_history`.

max_consecutive_failures = 30

url_timeout_total = 300.0

url_timeout_connect = None

url_error_retries = 10

url_error_delay_min = 0.5

url_error_delay_backoff = 1.5

url_error_delay_max = 5.0

maybe_trigger(*id: str = None*, **args: Any*, ***kwargs: Any*) → AsyncGenerator[Optional[faust.livecheck.models.TestExecution], None]
Schedule test execution, or not, based on probability setting.

Return type `AsyncGenerator[Optional[TestExecution], None]`

async trigger(*id: str = None*, **args: Any*, ***kwargs: Any*) → faust.livecheck.models.TestExecution
Schedule test execution ASAP.

Return type *TestExecution*

logger = <Logger faust.livecheck.case (WARNING)>

async run(**test_args: Any*, ***test_kwargs: Any*) → None
Override this to define your test case.

Return type `None`

async resolve_signal(*key: str*, *event: faust.livecheck.models.SignalEvent*) → None
Mark test execution signal as resolved.

Return type `None`

async execute(*test: faust.livecheck.models.TestExecution*) → None
Execute test using `TestRunner`.

Return type `None`

async on_test_start(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test starts executing.

Return type `None`

async on_test_skipped(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test is skipped.

Return type `None`

async on_test_failed(*runner: faust.livecheck.runners.TestRunner*, *exc: BaseException*) → None
Call when invariant in test execution fails.

Return type `None`

async on_test_error(*runner: faust.livecheck.runners.TestRunner*, *exc: BaseException*) → None
Call when a test execution raises an exception.

Return type `None`

async on_test_timeout(*runner: faust.livecheck.runners.TestRunner*, *exc: BaseException*) → None
Call when a test execution times out.

Return type None

async on_test_pass(*runner: faust.livecheck.runners.TestRunner*) → None
Call when a test execution passes.

Return type None

async post_report(*report: faust.livecheck.models.TestReport*) → None
Publish test report.

Return type None

async on_suite_fail(*exc: faust.livecheck.exceptions.SuiteFailed*, *new_state: faust.livecheck.models.State* = <*State.FAIL: 'FAIL'*>) → None
Call when the suite fails.

Return type None

property seconds_since_last_fail
Return number of seconds since any test failed. :rtype: Optional[float]

async get_url(*url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform GET request using HTTP client.

Return type Optional[bytes]

async post_url(*url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform POST request using HTTP client.

Return type Optional[bytes]

async url_request(*method: str, url: Union[str, yarl.URL], **kwargs: Any*) → Optional[bytes]
Perform URL request using HTTP client.

Return type Optional[bytes]

property current_test
Return the currently active test in this task (if any). :rtype: Optional[*TestExecution*]

property current_execution
Return the currently executing TestRunner in this task. :rtype: Optional[*TestRunner*]

property label
Return human-readable label for this test case. :rtype: str

faust.livecheck.exceptions

LiveCheck - related exceptions.

exception faust.livecheck.exceptions.**LiveCheckError**
Generic base class for LiveCheck test errors.

exception faust.livecheck.exceptions.**SuiteFailed**
The whole test suite failed (not just a test).

exception faust.livecheck.exceptions.**ServiceDown**
Suite failed after a depending service is not responding.

Used when for example a test case is periodically sending requests to a HTTP service, and that HTTP server is not responding.

exception faust.livecheck.exceptions.**SuiteStalled**
The suite is not running.

Raised when `warn_stalled_after=3600` is set and there has not been any execution requests in the last hour.

exception faust.livecheck.exceptions.**TestSkipped**
Test was skipped.

exception faust.livecheck.exceptions.**TestFailed**
The test failed an assertion.

exception faust.livecheck.exceptions.**TestRaised**
The test raised an exception.

exception faust.livecheck.exceptions.**TestTimeout**
The test timed out waiting for an event or during processing.

faust.livecheck.locals

Locals - Current test & execution context.

faust.livecheck.locals.**current_execution**() → Optional[faust.livecheck.locals._TestRunner]
Return the current *TestRunner*.

Return type Optional[_TestRunner]

faust.livecheck.locals.**current_test**() → Optional[faust.livecheck.models.TestExecution]
Return information about the current test (if any).

Return type Optional[*TestExecution*]

faust.livecheck.models

LiveCheck - Models.

class faust.livecheck.models.**State**
Test execution status.

INIT = 'INIT'

PASS = 'PASS'

FAIL = 'FAIL'

ERROR = 'ERROR'

TIMEOUT = 'TIMEOUT'

STALL = 'STALL'

SKIP = 'SKIP'

is_ok() → bool
Return `True` if this is considered an OK state.

Return type bool

class faust.livecheck.models.**SignalEvent**(*signal_name*, *case_name*, *key*, *value*, ***, *__strict__=True*, *__faust=None*, ***kwargs*) → None
Signal sent to test (see *faust.livecheck.signals.Signal*).

signal_name

case_name

key

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

value

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

asdict()

class faust.livecheck.models.**TestExecution**(*id*, *case_name*, *timestamp*, *test_args*, *test_kwargs*, *expires*, ***, *__strict__=True*, *__faust=None*, ***kwargs*) → None

Requested test execution.

id

case_name

timestamp

test_args

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

test_kwargs

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

expires

classmethod from_headers(*headers: Mapping*) → Optional[faust.livecheck.models.TestExecution]
Create instance from mapping of HTTP/Kafka headers.

Return type Optional[*TestExecution*]

as_headers() → Mapping
Return test metadata as mapping of HTTP/Kafka headers.

Return type Mapping[~KT, +VT_co]

ident
Return long identifier for this test used in logs.

shortident
Return short identifier for this test used in logs.

human_date
Return human-readable description of test timestamp.

was_issued_today
Return True if test was issued on todays date.

is_expired
Return True if this test already expired.

short_case_name
Return abbreviated case name.

asdict()

class faust.livecheck.models.**TestReport**(*case_name*, *state*, *signal_latency*, *test=None*, *runtime=None*, *error=None*, *traceback=None*, ***, *__strict__=True*, *__faust=None*, ***kwargs*) → None

Report after test execution.

case_name

state

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

test

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

runtime

signal_latency

error

traceback

asdict()

faust.livecheck.patches

Patches - LiveCheck integration with other frameworks/libraries.

faust.livecheck.patches.**patch_all**() → None
Apply all LiveCheck monkey patches.

Return type None

faust.livecheck.patches.aiohttp

LiveCheck aiohttp integration.

faust.livecheck.patches.aiohttp.**patch_all**() → None
Patch all aiohttp functions to integrate with LiveCheck.

Return type None

faust.livecheck.patches.aiohttp.**patch_aiohttp_session**() → None
Patch aiohttp.ClientSession to integrate with LiveCheck.

If there is any currently active test, we will use that to forward LiveCheck HTTP headers to the new HTTP request.

Return type None

class faust.livecheck.patches.aiohttp.**LiveCheckMiddleware**
LiveCheck support for aiohttp web servers.

This middleware is applied to all incoming web requests, and is used to extract LiveCheck HTTP headers.

If the web request is configured with the correct set of LiveCheck headers, we will use that to set the "current test" context.

faust.livecheck.runners

LiveCheck - Test runner.

class faust.livecheck.runners.**TestRunner**(*case: faust.livecheck.runners._Case*, *test: faust.livecheck.models.TestExecution*, *started: float*) → None

Execute and keep track of test instance.

state = 'INIT'

report = None

error = None

async execute() → None

Execute this test.

Return type None

async skip(*reason: str*) → NoReturn

Skip this test execution.

Return type _NoReturn

async on_skipped(*exc: faust.livecheck.exceptions.TestSkipped*) → None

Call when a test execution was skipped.

Return type None

async on_start() → None

Call when a test starts executing.

Return type None

async on_signal_wait(*signal: faust.livecheck.signals.BaseSignal*, *timeout: float*) → None

Call when the test is waiting for a signal.

Return type None

async on_signal_received(*signal: faust.livecheck.signals.BaseSignal*, *time_start: float*, *time_end: float*) → None

Call when a signal related to this test is received.

Return type None

async on_failed(*exc: BaseException*) → None

Call when an invariant in the test has failed.

Return type None

async on_error(*exc: BaseException*) → None

Call when test execution raises error.

Return type None

async on_timeout(*exc: BaseException*) → None

Call when test execution times out.

Return type None

async on_pass() → None

Call when test execution returns successfully.

Return type None

log_info(*msg: str*, **args: Any*) → None
Log information related to the current execution.

Return type None

end() → None
End test execution.

Return type None

faust.livecheck.signals

LiveCheck Signals - Test communication and synchronization.

class faust.livecheck.signals.**BaseSignal**(*name: str = ''*, *case: faust.livecheck.signals._Case = None*, *index: int = -1*) → None
Generic base class for signals.

async send(*value: VT = None*, ***, *key: Any = None*, *force: bool = False*) → None
Notify test that this signal is now complete.

Return type None

async wait(***, *key: Any = None*, *timeout: Union[datetime.timedelta, float, str] = None*) → VT
Wait for signal to be completed.

Return type ~VT

async resolve(*key: Any*, *event: faust.livecheck.models.SignalEvent*) → None
Resolve signal with value.

Return type None

clone(***kwargs: Any*) → faust.livecheck.signals.BaseSignal
Clone this signal using keyword arguments.

Return type *BaseSignal*[~VT]

class faust.livecheck.signals.**Signal**(*name: str = ''*, *case: faust.livecheck.signals._Case = None*, *index: int = -1*) → None
Signal for test case using Kafka.

Used to wait for something to happen elsewhere.

async send(*value: VT = None*, ***, *key: Any = None*, *force: bool = False*) → None
Notify test that this signal is now complete.

Return type None

async wait(***, *key: Any = None*, *timeout: Union[datetime.timedelta, float, str] = None*) → VT
Wait for signal to be completed.

Return type ~VT

1.6.6 Models

faust.models.base

Model descriptions.

The model describes the components of a data structure, kind of like a struct in C, but there's no limitation of what type of data structure the model is, or what it's used for.

A record (faust.models.record) is a model type that serialize into dictionaries, so the model describe the fields, and their types:

```
>>> class Point(Record):
...     x: int
...     y: int

>>> p = Point(10, 3)
>>> assert p.x == 10
>>> assert p.y == 3
>>> p
<Point: x=10, y=3>
>>> payload = p.dumps(serializer='json')
'{"x": 10, "y": 3, "__faust": {"ns": "__main__.Point"}}'
>>> p2 = Record.loads(payload)
>>> p2
<Point: x=10, y=3>
```

Models are mainly used for describing the data in messages: both keys and values can be described as models.

faust.models.base.**registry = {'@ClientAssignment': <class 'faust.assignor.client_assignmen**
Global map of namespace -> Model, used to find model classes by name. Every single model defined is added here automatically when a model class is defined.

faust.models.base.**maybe_model**(*arg: Any*) → Any
Convert argument to model if possible.

Return type Any

class faust.models.base.**Model**(**args: Any*, ***kwargs: Any*) → None
Meta description model for serialization.

classmethod loads(*s: bytes*, ***, *default_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → faust.types.models.ModelT
Deserialize model object from bytes.

Keyword Arguments **serializer** (*CodecArg*) – Default serializer to use if no custom serializer was set for this model subclass.

Return type *ModelT*

classmethod make_final() → None

Return type None

abstract to_representation() → Any
Convert object to JSON serializable object.

Return type Any

is_valid() → bool

Return type bool

validate() → List[faust.exceptions.ValidationError]

Return type List[ValidationError]

validate_or_raise() → None

Return type None

property validation_errors

Return type List[ValidationError]

derive(**objects: faust.types.models.ModelT*, ***fields: Any*) → faust.types.models.ModelT
Derive new model with certain fields changed.

Return type *ModelT*

dumps(*, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → bytes
Serialize object to the target serialization format.

Return type bytes

faust.models.fields

class faust.models.fields.**FieldDescriptor**(*, *field: str = None*, *input_name: str = None*, *output_name: str = None*, *type: Type[T] = None*, *model: Type[faust.types.models.ModelT] = None*, *required: bool = True*, *default: T = None*, *parent: faust.types.models.FieldDescriptorT = None*, *coerce: bool = None*, *generic_type: Type = None*, *member_type: Type = None*, *exclude: bool = None*, *date_parser: Callable[Any, datetime.datetime] = None*, ***options: Any*) → None

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

field = None
Name of attribute on Model.

input_name = None
Name of field in serialized data (if differs from *field*). Defaults to *field*.

output_name = None
Name of field when serializing data (if differs from *field*). Defaults to *input_name*.

type = None
Type of value (e.g. int, or Optional[int]).

model = None
The model class this field is associated with.

required = True
Set if a value for this field is required (cannot be None).

default = None
Default value for non-required field.

generic_type = None
If this holds a generic type such as list/set/dict then this holds the type of collection.

coerce = False
Coerce value to field descriptors type. This means assigning a value to this field, will first convert the value to the requested type. For example for a *FloatField* the input will be converted to float, and passing any value that cannot be converted to float will raise an error.

If coerce is not enabled you can store any type of value.

Note: None is usually considered a valid value for any field but this depends on the descriptor type.

exclude = False
Exclude field from model representation. This means the field will not be part of the serialized structure. (Model.dumps(), Model.asdict(), and Model.to_representation()).

clone(***kwargs: Any*) → faust.types.models.FieldDescriptorT

Return type *FieldDescriptorT*[~T]

as_dict() → Mapping[str, Any]

Return type Mapping[str, Any]

validate_all(*value: Any*) → Iterable[faust.exceptions.ValidationError]

Return type Iterable[ValidationError]

validate(*value: T*) → Iterable[faust.exceptions.ValidationError]

Return type Iterable[ValidationError]

prepare_value(*value: Any*, *, *coerce: bool = None*) → Optional[T]

Return type Optional[~T]

should_coerce(*value: Any*, *coerce: bool = None*) → bool

Return type bool

getattr(*obj: faust.types.models.ModelT*) → T
Get attribute from model recursively.

Supports recursive lookups e.g. `model.getattr('x.y.z')`.

Return type ~T

validation_error(*reason: str*) → faust.exceptions.ValidationError

Return type `ValidationError`

property ident
Return the fields identifier. :rtype: `str`

class faust.models.fields.**NumberField**(*, *max_value: int = None*, *min_value: int = None*, ***kwargs: Any*) → None

validate(*value: T*) → Iterable[faust.exceptions.ValidationError]

Return type `Iterable`[`ValidationError`]

class faust.models.fields.**IntegerField**(*, *max_value: int = None*, *min_value: int = None*, ***kwargs: Any*) → None

prepare_value(*value: Any*, *, *coerce: bool = None*) → Optional[int]

Return type `Optional`[`int`]

class faust.models.fields.**FloatField**(*, *max_value: int = None*, *min_value: int = None*, ***kwargs: Any*) → None

prepare_value(*value: Any*, *, *coerce: bool = None*) → Optional[float]

Return type `Optional`[`float`]

class faust.models.fields.**DecimalField**(*, *max_digits: int = None*, *max_decimal_places: int = None*, ***kwargs: Any*) → None

max_digits = None

max_decimal_places = None

prepare_value(*value: Any*, *, *coerce: bool = None*) → Optional[decimal.Decimal]

Return type `Optional`[`Decimal`]

validate(*value: decimal.Decimal*) → Iterable[faust.exceptions.ValidationError]

Return type `Iterable`[`ValidationError`]

class faust.models.fields.**StringField**(*, *max_length: int = None*, *min_length: int = None*, *trim_whitespace: bool = False*, *allow_blank: bool = False*, ***kwargs: Any*) → None

prepare_value(*value: Any*, *, *coerce: bool = None*) → Optional[str]

Return type `Optional`[`str`]

class faust.models.fields.**DatetimeField**(*, *field: str = None*, *input_name: str = None*, *output_name: str = None*, *type: Type[T] = None*, *model: Type[faust.types.models.ModelT] = None*, *required: bool = True*, *default: T = None*, *parent: faust.types.models.FieldDescriptorT = None*, *coerce: bool = None*, *generic_type: Type = None*, *member_type: Type = None*, *exclude: bool = None*, *date_parser: Callable[Any, datetime.datetime] = None*, ***options: Any*) → None

prepare_value(*value: Any*, ***, *coerce: bool = None*) → Optional[datetime.datetime]

Return type Optional[datetime]

class faust.models.fields.**BytesField**(*, *encoding: str = None*, *errors: str = None*, ***kwargs: Any*) → None

encoding = 'utf-8'

errors = 'strict'

prepare_value(*value: Any*, ***, *coerce: bool = None*) → Optional[bytes]

Return type Optional[bytes]

faust.models.fields.**field_for_type**(*typ: Type*) → Type[faust.types.models.FieldDescriptorT]

Return type Type[*FieldDescriptorT*[~T]]

faust.models.record

Record - Dictionary Model.

class faust.models.record.**Record** → None

Describes a model type that is a record (Mapping).

Examples

```
>>> class LogEvent(Record, serializer='json'):
...     severity: str
...     message: str
...     timestamp: float
...     optional_field: str = 'default value'
```

```
>>> event = LogEvent(
...     severity='error',
...     message='Broken pact',
...     timestamp=666.0,
... )
```

```
>>> event.severity
'error'
```

```
>>> serialized = event.dumps()
'{"severity": "error", "message": "Broken pact", "timestamp": 666.0}'
```

```
>>> restored = LogEvent.loads(serialized)
<LogEvent: severity='error', message='Broken pact', timestamp=666.0>
```

```
>>> # You can also subclass a Record to create a new record
>>> # with additional fields:
>>> class RemoteLogEvent(LogEvent):
...     url: str
```

```
>>> # You can also refer to record fields and pass them around:
>>> LogEvent.severity
>>> <FieldDescriptor: LogEvent.severity (str)>
```

classmethod from_data(*data: Mapping*, ***, *preferred_type: Type[faust.types.models.ModelT] = None*) → faust.models.record.Record
Create model object from Python dictionary.

Return type *Record*

to_representation() → Mapping[str, Any]
Convert model to its Python generic counterpart.

Records will be converted to dictionary.

Return type Mapping[str, Any]

asdict() → Dict[str, Any]
Convert record to Python dictionary.

Return type Dict[str, Any]

1.6.7 Sensors

faust.sensors

Sensors.

class faust.sensors.**Sensor**(***, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None
Base class for sensors.

This sensor does not do anything at all, but can be subclassed to create new monitors.

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Message received by a consumer.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Message sent to a stream as an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Event was acknowledged by stream.

Notes

Acknowledged means a stream finished processing the event, but given that multiple streams may be handling the same event, the message cannot be committed before all streams have processed it. When all streams have acknowledged the event, it will go through `on_message_out()` just before offsets are committed.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
All streams finished processing message.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None
Topic buffer full so conductor had to wait.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Key retrieved from table.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Value set for key in table.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Key deleted from table.

Return type None

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Consumer is about to commit topic offset.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Consumer finished committing topic offset.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
About to send a message.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Message successfully sent.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Error while sending message.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, ***, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT, request: faust.web.base.Request, response: Optional[faust.web.base.Response], state: Dict, *, view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

asdict() → Mapping
Convert sensor state to dictionary.

Return type Mapping[~KT, +VT_co]

logger = <Logger faust.sensors.base (WARNING)>

class faust.sensors.**SensorDelegate**(*app: faust.types.app.AppT*) → None
A class that delegates sensor methods to a list of sensors.

add(*sensor: faust.types.sensors.SensorT*) → None
Add sensor.

Return type None

remove(*sensor: faust.types.sensors.SensorT*) → None
Remove sensor.

Return type None

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call before message is delegated to streams.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Call when stream starts processing an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Call when stream is done processing an event.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None
Call when conductor topic buffer is full and has to wait.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call when message is fully acknowledged and can be committed.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when value in table is retrieved.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Call when new value for key in table is set.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when key in a table is deleted.

Return type None

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Call when consumer commit offset operation starts.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Call when consumer commit offset operation completed.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
Call when message added to producer buffer.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Call when producer finished sending message.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Call when producer was unable to publish message.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, ***, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT, request: faust.web.base.Request, response: Optional[faust.web.base.Response], state: Dict, *, view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

class faust.sensors.**Monitor**(***, *max_avg_history: int = None*, *max_commit_latency_history: int = None*, *max_send_latency_history: int = None*, *max_assignment_latency_history: int = None*, *messages_sent: int = 0*, *tables: MutableMapping[str, faust.sensors.monitor.TableState] = None*, *messages_active: int = 0*, *events_active: int = 0*, *messages_received_total: int = 0*, *messages_received_by_topic: Counter[str] = None*, *events_total: int = 0*, *events_by_stream: Counter[faust.types.streams.StreamT] = None*, *events_by_task: Counter[_asyncio.Task] = None*, *events_runtime: Deque[float] = None*, *commit_latency: Deque[float] = None*, *send_latency: Deque[float] = None*, *assignment_latency: Deque[float] = None*, *events_s: int = 0*, *messages_s: int = 0*, *events_runtime_avg: float = 0.0*, *topic_buffer_full: Counter[faust.types.topics.TopicT] = None*, *rebalances: int = None*, *rebalance_return_latency: Deque[float] = None*, *rebalance_end_latency: Deque[float] = None*, *rebalance_return_avg: float = 0.0*, *rebalance_end_avg: float = 0.0*, *time: Callable[float] = <built-in function monotonic>*, *http_response_codes: Counter[http.HTTPStatus] = None*, *http_response_latency: Deque[float] = None*, *http_response_latency_avg: float = 0.0*, ***kwargs: Any*) → None
Default Faust Sensor.

This is the default sensor, recording statistics about events, etc.

send_errors = 0
Number of produce operations that ended in error.

assignments_completed = 0
Number of partition assignments completed.

assignments_failed = 0
Number of partitions assignments that failed.

max_avg_history = 100
Max number of total run time values to keep to build average.

max_commit_latency_history = 30
Max number of commit latency numbers to keep.

max_send_latency_history = 30
Max number of send latency numbers to keep.

max_assignment_latency_history = 30
Max number of assignment latency numbers to keep.

rebalances = 0
Number of rebalances seen by this worker.

tables = None
Mapping of tables

commit_latency = None
Deque of commit latency values

send_latency = None
Deque of send latency values

assignment_latency = None
Deque of assignment latency values.

rebalance_return_latency = None
Deque of previous n rebalance return latencies.

rebalance_end_latency = None
Deque of previous n rebalance end latencies.

rebalance_return_avg = 0.0
Average rebalance return latency.

rebalance_end_avg = 0.0
Average rebalance end latency.

messages_active = 0
Number of messages currently being processed.

messages_received_total = 0
Number of messages processed in total.

messages_received_by_topic = None
Count of messages received by topic

messages_sent = 0
Number of messages sent in total.

messages_sent_by_topic = None
Number of messages sent by topic.

messages_s = 0
Number of messages being processed this second.

events_active = 0
Number of events currently being processed.

events_total = 0
Number of events processed in total.

events_by_task = None
Count of events processed by task

events_by_stream = None
Count of events processed by stream

events_s = 0
Number of events being processed this second.

events_runtime_avg = 0.0
Average event runtime over the last second.

events_runtime = None
Deque of run times used for averages

topic_buffer_full = None
Counter of times a topics buffer was full

http_response_codes = None
Counter of returned HTTP status codes.

http_response_latency = None
Deque of previous n HTTP request->response latencies.

http_response_latency_avg = 0.0
Average request->response latency.

metric_counts = None
Arbitrary counts added by apps

tp_committed_offsets = None
Last committed offsets by TopicPartition

tp_read_offsets = None
Last read offsets by TopicPartition

tp_end_offsets = None
Log end offsets by TopicPartition

secs_since(*start_time: float*) → float
Given timestamp start, return number of seconds since that time.

Return type float

ms_since(*start_time: float*) → float
Given timestamp start, return number of ms since that time.

Return type float

logger = <Logger faust.sensors.monitor (WARNING)>

secs_to_ms(*timestamp: float*) → float
Convert seconds to milliseconds.

Return type float

asdict() → Mapping
Return monitor state as dictionary.

Return type Mapping[~KT, +VT_co]

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call before message is delegated to streams.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Call when stream starts processing an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Call when stream is done processing an event.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None
Call when conductor topic buffer is full and has to wait.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call when message is fully acknowledged and can be committed.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when value in table is retrieved.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Call when new value for key in table is set.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when key in a table is deleted.

Return type None

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Consumer is about to commit topic offset.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Call when consumer commit offset operation completed.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
Call when message added to producer buffer.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Call when producer finished sending message.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Call when producer was unable to publish message.

Return type None

count(*metric_name: str*, *count: int = 1*) → None
Count metric by name.

Return type None

on_tp_commit(*tp_offsets: MutableMapping[faust.types.tuples.TP, int]*) → None
Call when offset in topic partition is committed.

Return type None

track_tp_end_offset(*tp: faust.types.tuples.TP*, *offset: int*) → None
Track new topic partition end offset for monitoring lags.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, ***, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT, request: faust.web.base.Request, response: Optional[faust.web.base.Response], state: Dict, *, view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

class faust.sensors.**TableState**(*table: faust.types.tables.CollectionT*, ***, *keys_retrieved: int = 0*, *keys_updated: int = 0*, *keys_deleted: int = 0*) → None

Represents the current state of a table.

table = None

keys_retrieved = 0

Number of times a key has been retrieved from this table.

keys_updated = 0

Number of times a key has been created/changed in this table.

keys_deleted = 0

Number of times a key has been deleted from this table.

asdict() → Mapping

Return table state as dictionary.

Return type Mapping[~KT, +VT_co]

faust.sensors.base

Base-interface for sensors.

class faust.sensors.base.**Sensor**(***, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None

Base class for sensors.

This sensor does not do anything at all, but can be subclassed to create new monitors.

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None

Message received by a consumer.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]

Message sent to a stream as an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None

Event was acknowledged by stream.

Notes

Acknowledged means a stream finished processing the event, but given that multiple streams may be handling the same event, the message cannot be committed before all streams have processed it. When all streams have acknowledged the event, it will go through *on_message_out()* just before offsets are committed.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None

All streams finished processing message.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None

Topic buffer full so conductor had to wait.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Key retrieved from table.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Value set for key in table.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Key deleted from table.

Return type None

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Consumer is about to commit topic offset.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Consumer finished committing topic offset.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
About to send a message.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Message successfully sent.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Error while sending message.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, *, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT, request: faust.web.base.Request, response: Optional[faust.web.base.Response], state: Dict, *, view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

asdict() → Mapping
Convert sensor state to dictionary.

Return type Mapping[~KT, +VT_co]

logger = <Logger faust.sensors.base (WARNING)>

class faust.sensors.base.**SensorDelegate**(*app: faust.types.app.AppT*) → None
A class that delegates sensor methods to a list of sensors.

add(*sensor: faust.types.sensors.SensorT*) → None
Add sensor.

Return type None

remove(*sensor: faust.types.sensors.SensorT*) → None
Remove sensor.

Return type None

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call before message is delegated to streams.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Call when stream starts processing an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Call when stream is done processing an event.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None
Call when conductor topic buffer is full and has to wait.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call when message is fully acknowledged and can be committed.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when value in table is retrieved.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Call when new value for key in table is set.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when key in a table is deleted.

Return type None

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Call when consumer commit offset operation starts.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Call when consumer commit offset operation completed.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
Call when message added to producer buffer.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Call when producer finished sending message.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Call when producer was unable to publish message.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, ***, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, *response: Optional[faust.web.base.Response]*, *state: Dict*, ***, *view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

faust.sensors.datadog

Monitor using datadog.

class faust.sensors.datadog.**DatadogMonitor**(*host: str = 'localhost'*, *port: int = 8125*, *prefix: str = 'faust-app'*, *rate: float = 1.0*, ***kwargs: Any*) → None
Datadog Faust Sensor.

This sensor, records statistics to datadog agents along with computing metrics for the stats server

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call before message is delegated to streams.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Call when stream starts processing an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Call when stream is done processing an event.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call when message is fully acknowledged and can be committed.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when value in table is retrieved.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Call when new value for key in table is set.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when key in a table is deleted.

Return type None

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Call when consumer commit offset operation completed.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
Call when message added to producer buffer.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Call when producer finished sending message.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Call when producer was unable to publish message.

Return type None

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

count(*metric_name: str*, *count: int = 1*) → None
Count metric by name.

Return type None

on_tp_commit(*tp_offsets: MutableMapping[faust.types.tuples.TP, int]*) → None
Call when offset in topic partition is committed.

Return type None

track_tp_end_offset(*tp: faust.types.tuples.TP*, *offset: int*) → None
Track new topic partition end offset for monitoring lags.

Return type None

on_web_request_end(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, *response: Optional[faust.web.base.Response]*, *state: Dict*, ***, *view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

logger = <Logger faust.sensors.datadog (WARNING)>

client
Return the datadog client.

faust.sensors.monitor

Monitor - sensor tracking metrics.

class faust.sensors.monitor.**TableState**(*table: faust.types.tables.CollectionT*, ***, *keys_retrieved: int = 0*, *keys_updated: int = 0*, *keys_deleted: int = 0*) → None

Represents the current state of a table.

table = None

keys_retrieved = 0
Number of times a key has been retrieved from this table.

keys_updated = 0
Number of times a key has been created/changed in this table.

keys_deleted = 0
Number of times a key has been deleted from this table.

asdict() → Mapping
Return table state as dictionary.

Return type Mapping[~KT, +VT_co]

class faust.sensors.monitor.**Monitor**(***, *max_avg_history: int = None*, *max_commit_latency_history: int = None*, *max_send_latency_history: int = None*, *max_assignment_latency_history: int = None*, *messages_sent: int = 0*, *tables: MutableMapping[str, faust.sensors.monitor.TableState] = None*, *messages_active: int = 0*, *events_active: int = 0*, *messages_received_total: int = 0*, *messages_received_by_topic: Counter[str] = None*, *events_total: int = 0*, *events_by_stream: Counter[faust.types.streams.StreamT] = None*, *events_by_task: Counter[_asyncio.Task] = None*, *events_runtime: Deque[float] = None*, *commit_latency: Deque[float] = None*, *send_latency: Deque[float] = None*, *assignment_latency: Deque[float] = None*, *events_s: int = 0*, *messages_s: int = 0*, *events_runtime_avg: float = 0.0*, *topic_buffer_full: Counter[faust.types.topics.TopicT] = None*, *rebalances: int = None*, *rebalance_return_latency: Deque[float] = None*, *rebalance_end_latency: Deque[float] = None*, *rebalance_return_avg: float = 0.0*, *rebalance_end_avg: float = 0.0*, *time: Callable[float] = <built-in function monotonic>*, *http_response_codes: Counter[http.HTTPStatus] = None*, *http_response_latency: Deque[float] = None*, *http_response_latency_avg: float = 0.0*, ***kwargs: Any*) → None

Default Faust Sensor.

This is the default sensor, recording statistics about events, etc.

send_errors = 0
Number of produce operations that ended in error.

assignments_completed = 0
Number of partition assignments completed.

assignments_failed = 0
Number of partitions assignments that failed.

max_avg_history = 100
Max number of total run time values to keep to build average.

max_commit_latency_history = 30
Max number of commit latency numbers to keep.

max_send_latency_history = 30
Max number of send latency numbers to keep.

max_assignment_latency_history = 30
Max number of assignment latency numbers to keep.

rebalances = 0
Number of rebalances seen by this worker.

tables = None
Mapping of tables

commit_latency = None
Deque of commit latency values

send_latency = None
Deque of send latency values

assignment_latency = None
Deque of assignment latency values.

rebalance_return_latency = None
Deque of previous n rebalance return latencies.

rebalance_end_latency = None
Deque of previous n rebalance end latencies.

rebalance_return_avg = 0.0
Average rebalance return latency.

rebalance_end_avg = 0.0
Average rebalance end latency.

messages_active = 0
Number of messages currently being processed.

messages_received_total = 0
Number of messages processed in total.

messages_received_by_topic = None
Count of messages received by topic

messages_sent = 0
Number of messages sent in total.

messages_sent_by_topic = None
Number of messages sent by topic.

messages_s = 0
Number of messages being processed this second.

events_active = 0
Number of events currently being processed.

events_total = 0
Number of events processed in total.

events_by_task = None
Count of events processed by task

events_by_stream = None
Count of events processed by stream

events_s = 0
Number of events being processed this second.

events_runtime_avg = 0.0
Average event runtime over the last second.

events_runtime = None
Deque of run times used for averages

topic_buffer_full = None
Counter of times a topics buffer was full

http_response_codes = None
Counter of returned HTTP status codes.

http_response_latency = None
Deque of previous n HTTP request->response latencies.

http_response_latency_avg = 0.0
Average request->response latency.

metric_counts = None
Arbitrary counts added by apps

tp_committed_offsets = None
Last committed offsets by TopicPartition

tp_read_offsets = None
Last read offsets by TopicPartition

tp_end_offsets = None
Log end offsets by TopicPartition

secs_since(*start_time: float*) → float
Given timestamp start, return number of seconds since that time.

Return type float

ms_since(*start_time: float*) → float
Given timestamp start, return number of ms since that time.

Return type float

logger = <Logger faust.sensors.monitor (WARNING)>

secs_to_ms(*timestamp: float*) → float
Convert seconds to milliseconds.

Return type float

asdict() → Mapping
Return monitor state as dictionary.

Return type Mapping[~KT, +VT_co]

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call before message is delegated to streams.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Call when stream starts processing an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Call when stream is done processing an event.

Return type None

on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None
Call when conductor topic buffer is full and has to wait.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call when message is fully acknowledged and can be committed.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when value in table is retrieved.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Call when new value for key in table is set.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when key in a table is deleted.

Return type None

on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any
Consumer is about to commit topic offset.

Return type Any

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Call when consumer commit offset operation completed.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
Call when message added to producer buffer.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Call when producer finished sending message.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Call when producer was unable to publish message.

Return type None

count(*metric_name: str*, *count: int = 1*) → None
Count metric by name.

Return type None

on_tp_commit(*tp_offsets: MutableMapping[faust.types.tuples.TP, int]*) → None
Call when offset in topic partition is committed.

Return type None

track_tp_end_offset(*tp: faust.types.tuples.TP*, *offset: int*) → None
Track new topic partition end offset for monitoring lags.

Return type None

on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict
Partition assignor is starting to assign partitions.

Return type Dict[~KT, ~VT]

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

on_web_request_start(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, ***, *view: faust.web.views.View = None*) → Dict
Web server started working on request.

Return type Dict[~KT, ~VT]

on_web_request_end(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, *response: Optional[faust.web.base.Response]*, *state: Dict*, ***, *view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

faust.sensors.statsd

Monitor using Statsd.

class faust.sensors.statsd.**StatsdMonitor**(*host: str = 'localhost'*, *port: int = 8125*, *prefix: str = 'faust-app'*, *rate: float = 1.0*, ***kwargs: Any*) → None

Statsd Faust Sensor.

This sensor, records statistics to Statsd along with computing metrics for the stats server

on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call before message is delegated to streams.

Return type None

on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]
Call when stream starts processing an event.

Return type Optional[Dict[~KT, ~VT]]

on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None
Call when stream is done processing an event.

Return type None

on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None
Call when message is fully acknowledged and can be committed.

Return type None

on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when value in table is retrieved.

Return type None

on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None
Call when new value for key in table is set.

Return type None

on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None
Call when key in a table is deleted.

Return type None

on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None
Call when consumer commit offset operation completed.

Return type None

on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any
Call when message added to producer buffer.

Return type Any

on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None
Call when producer finished sending message.

Return type None

on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None
Call when producer was unable to publish message.

Return type None

on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None
Partition assignor did not complete assignor due to error.

Return type None

on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None
Partition assignor completed assignment.

Return type None

on_rebalance_start(*app: faust.types.app.AppT*) → Dict
Cluster rebalance in progress.

Return type Dict[~KT, ~VT]

on_rebalance_return(*app: faust.types.app.AppT*, *state: Dict*) → None
Consumer replied assignment is done to broker.

Return type None

on_rebalance_end(*app: faust.types.app.AppT*, *state: Dict*) → None
Cluster rebalance fully completed (including recovery).

Return type None

count(*metric_name: str*, *count: int = 1*) → None
Count metric by name.

Return type None

on_tp_commit(*tp_offsets: MutableMapping[faust.types.tuples.TP, int]*) → None
Call when offset in topic partition is committed.

Return type None

logger = <Logger faust.sensors.statsd (WARNING)>

track_tp_end_offset(*tp: faust.types.tuples.TP*, *offset: int*) → None
Track new topic partition end offset for monitoring lags.

Return type None

on_web_request_end(*app: faust.types.app.AppT*, *request: faust.web.base.Request*, *response: Optional[faust.web.base.Response]*, *state: Dict*, ***, *view: faust.web.views.View = None*) → None
Web server finished working on request.

Return type None

client
Return statsd client.

1.6.8 Serializers

faust.serializers.codecs

Serialization utilities.

Supported codecs

- **raw** - No encoding/serialization (bytes only).
- **json** - json with UTF-8 encoding.
- **yaml** - YAML (safe version)
- **pickle** - pickle with base64 encoding (not urlsafe).
- **binary** - base64 encoding (not urlsafe).

Serialization by name

The *dumps()* function takes a codec name and the object to encode, then returns bytes:

```
>>> s = dumps('json', obj)
```

For the reverse direction, the *loads()* function takes a codec name and bytes to decode:

```
>>> obj = loads('json', s)
```

You can also combine encoders in the name, like in this case where json is combined with gzip compression:

```
>>> obj = loads('json|gzip', s)
```

Codec registry

Codecs are configured by name and this module maintains a mapping from name to *Codec* instance: the `codecs` attribute.

You can add a new codec to this mapping by:

```
>>> from faust.serializers import codecs
>>> codecs.register(custom, custom_serializer())
```

A codec subclass requires two methods to be implemented: `_loads()` and `_dumps()`:

```
import msgpack

from faust.serializers import codecs

class raw_msgpack(codecs.Codec):

    def _dumps(self, obj: Any) -> bytes:
        return msgpack.dumps(obj)

    def _loads(self, s: bytes) -> Any:
        return msgpack.loads(s)
```

Our codec now encodes/decodes to raw msgpack format, but we may also need to transfer this payload over a transport easily confused by binary data, such as JSON where everything is Unicode.

You can chain codecs together, so to add a binary text encoding like Base64, to your codec, we use the `|` operator to form a combined codec:

```
def msgpack() -> codecs.Codec:
    return raw_msgpack() | codecs.binary()

codecs.register('msgpack', msgpack())
```

At this point we monkey-patched Faust to support our codec, and we can use it to define records like this:

```
>>> from faust.serializers import Record
>>> class Point(Record, serializer='msgpack'):
...     x: int
...     y: int
```

The problem with monkey-patching is that we must make sure the patching happens before we use the feature.

Faust also supports registering *codec extensions* using setuptools entry points, so instead we can create an installable msgpack extension.

To do so we need to define a package with the following directory layout:

```
faust-msgpack/
    setup.py
    faust_msgpack.py
```

The first file, `faust-msgpack/setup.py`, defines metadata about our package and should look like the following example:

```
from setuptools import setup, find_packages

setup(
    name='faust-msgpack',
    version='1.0.0',
    description='Faust msgpack serialization support',
    author='Ola A. Normann',
    author_email='ola@normann.no',
    url='http://github.com/example/faust-msgpack',
    platforms=['any'],
    license='BSD',
    packages=find_packages(exclude=['ez_setup', 'tests', 'tests.*']),
    zip_safe=False,
    install_requires=['msgpack-python'],
```

(continues on next page)

(continued from previous page)

```
    tests_require=[],
    entry_points={
        'faust.codecs': [
            'msgpack = faust_msgpack:msgpack',
        ],
    },
)
```

The most important part being the `entry_points` key which tells Faust how to load our plugin. We have set the name of our codec to `msgpack` and the path to the codec class to be `faust_msgpack:msgpack`. This will be imported by Faust as `from faust_msgpack import msgpack`, so we need to define that part next in our `faust-msgpack/faust_msgpack.py` module:

```
from faust.serializers import codecs

class raw_msgpack(codecs.Codec):

    def _dumps(self, obj: Any) -> bytes:
        return msgpack.dumps(s)


def msgpack() -> codecs.Codec:
    return raw_msgpack() | codecs.binary()
```

That's it! To install and use our new extension we do:

```
$ python setup.py install
```

At this point may want to publish this on PyPI to share the extension with other Faust users.

class faust.serializers.codecs.**Codec**(*children: Tuple[faust.types.codecs.CodecT, ...] = None*, ***kwargs: Any*) → None

Base class for codecs.

children = None

next steps in the recursive codec chain. `x = pickle | binary` returns codec with children set to `(pickle, binary)`.

nodes = None

cached version of children including this codec as the first node. could use chain below, but seems premature so just copying the list.

kwargs = None

subclasses can support keyword arguments, the base implementation of *clone()* uses this to preserve keyword arguments in copies.

dumps(*obj: Any*) → bytes

Encode object `obj`.

Return type bytes

loads(*s: bytes*) → Any

Decode object from string.

Return type Any

clone(**children: faust.types.codecs.CodecT*) → faust.types.codecs.CodecT

Create a clone of this codec, with optional children added.

Return type *CodecT*

faust.serializers.codecs.**register**(*name: str*, *codec: faust.types.codecs.CodecT*) → None
Register new codec in the codec registry.

Return type None

faust.serializers.codecs.**get_codec**(*name_or_codec: Union[faust.types.codecs.CodecT, str, None]*) → faust.types.codecs.CodecT
Get codec by name.

Return type *CodecT*

faust.serializers.codecs.**dumps**(*codec: Union[faust.types.codecs.CodecT, str, None]*, *obj: Any*) → bytes
Encode object into bytes.

Return type bytes

faust.serializers.codecs.**loads**(*codec: Union[faust.types.codecs.CodecT, str, None]*, *s: bytes*) → Any
Decode object from bytes.

Return type Any

faust.serializers.registry

Registry of supported codecs (serializers, compressors, etc.).

class faust.serializers.registry.**Registry**(*key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = 'json'*) → None
Serializing message keys/values.

Parameters

- **key_serializer** (Union[*CodecT*, str, None]) – Default key serializer to use when none provided.
- **value_serializer** (Union[*CodecT*, str, None]) – Default value serializer to use when none provided.

loads_key(*typ: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str], None]*, *key: Optional[bytes]*, ***, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → Union[bytes, faust.types.core._ModelT, Any, None]
Deserialize message key.

Parameters

- **typ** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – Model to use for deserialization.
- **key** (Optional[bytes]) – Serialized key.
- **serializer** (Union[*CodecT*, str, None]) – Codec to use for this value. If not set the default will be used (key_serializer).

Return type Union[bytes, _ModelT, Any, None]

loads_value(*typ: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str], None]*, *value: Optional[bytes]*, ***, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → Any
Deserialize value.

Parameters

- **typ** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – Model to use for deserialization.
- **value** (Optional[bytes]) – bytes to deserialize.
- **serializer** (Union[*CodecT*, str, None]) – Codec to use for this value. If not set the default will be used (value_serializer).

Return type Any

dumps_key(*typ: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str], None], key: Union[bytes, faust.types.core._ModelT, Any, None], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None, skip: Tuple[Type, ...] = (<class 'bytes'>,)*) → Optional[bytes]

Serialize key.

Parameters

- **typ** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – Model hint (can also be str/bytes). When typ=str or bytes, raw serializer is assumed.
- **key** (Union[bytes, _ModelT, Any, None]) – The key value to serializer.
- **serializer** (Union[*CodecT*, str, None]) – Codec to use for this key, if it is not a model type. If not set the default will be used (key_serializer).

Return type Optional[bytes]

dumps_value(*typ: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str], None], value: Union[bytes, faust.types.core._ModelT, Any], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None, skip: Tuple[Type, ...] = (<class 'bytes'>,)*) → Optional[bytes]

Serialize value.

Parameters

- **typ** (Union[Type[*ModelT*], Type[bytes], Type[str], None]) – Model hint (can also be str/bytes). When typ=str or bytes, raw serializer is assumed.
- **key** – The value to serializer.
- **serializer** (Union[*CodecT*, str, None]) – Codec to use for this value, if it is not a model type. If not set the default will be used (value_serializer).

Return type Optional[bytes]

Model

Return the faust.Model class used by this serializer.

faust.serializers.schemas

class faust.serializers.schemas.**Schema**(**, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, allow_empty: bool = None*) → None

update(*, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, allow_empty: bool = None) → None

Return type None

loads_key(app: faust.types.app.AppT, message: faust.types.tuples.Message, *, loads: Callable = None, serializer: Union[faust.types.codecs.CodecT, str, None] = None) → KT

Return type ~KT

loads_value(app: faust.types.app.AppT, message: faust.types.tuples.Message, *, loads: Callable = None, serializer: Union[faust.types.codecs.CodecT, str, None] = None) → VT

Return type ~VT

dumps_key(app: faust.types.app.AppT, key: Union[bytes, faust.types.core._ModelT, Any, None], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

dumps_value(app: faust.types.app.AppT, value: Union[bytes, faust.types.core._ModelT, Any], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

on_dumps_key_prepare_headers(key: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]) → Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

Return type Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

on_dumps_value_prepare_headers(value: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]) → Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

Return type Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

async decode(app: faust.types.app.AppT, message: faust.types.tuples.Message, *, propagate: bool = False) → faust.types.events.EventT

Decode message from topic (compiled function not cached).

Return type *EventT*[]

compile(app: faust.types.app.AppT, *, on_key_decode_error: Callable[[Exception, faust.types.tuples.Message], Awaitable[None]] = <function _noop_decode_error>, on_value_decode_error: Callable[[Exception, faust.types.tuples.Message], Awaitable[None]] = <function _noop_decode_error>, default_propagate: bool = False) → Callable[..., Awaitable[faust.types.events.EventT]]

Compile function used to decode event.

Return type Callable[…, Awaitable[*EventT*[]]]

1.6.9 Stores

faust.stores

Storage registry.

faust.stores.**by_name**(*name: Union[_T, str]*) → _T

Return type ~_T

faust.stores.**by_url**(*url: Union[str, yarl.URL]*) → _T
Get class associated with URL (scheme is used as alias key).

Return type ~_T

faust.stores.base

Base class for table storage drivers.

class faust.stores.base.**Store**(*url: Union[str, yarl.URL], app: faust.types.app.AppT, table: faust.types.tables.CollectionT, *, table_name: str = '', key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, options: Mapping[str, Any] = None, **kwargs: Any*) → None
Base class for table storage drivers.

persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]
Return the persisted offset for this topic and partition.

Return type Optional[int]

set_persisted_offset(*tp: faust.types.tuples.TP, offset: int*) → None
Set the persisted offset for this topic and partition.

Return type None

async need_active_standby_for(*tp: faust.types.tuples.TP*) → bool
Return True if we have a copy of standby from elsewhere.

Return type bool

async on_rebalance(*table: faust.types.tables.CollectionT, assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None
Handle rebalancing of the cluster.

Return type None

async on_recovery_completed(*active_tps: Set[faust.types.tuples.TP], standby_tps: Set[faust.types.tuples.TP]*) → None
Signal that table recovery completed.

Return type None

property label
Return short description of this store. :rtype: str

logger = <Logger faust.stores.base (WARNING)>

class faust.stores.base.**SerializedStore**(*url: Union[str, yarl.URL], app: faust.types.app.AppT, table: faust.types.tables.CollectionT, *, table_name: str = '', key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, options: Mapping[str, Any] = None, **kwargs: Any*) → None

Base class for table storage drivers requiring serialization.

apply_changelog_batch(*batch: Iterable[faust.types.events.EventT], to_key: Callable[Any, KT], to_value: Callable[Any, VT]*) → None

Apply batch of events from changelog topic to this store.

Return type None

keys() → collections.abc.KeysView

Return view of keys in the K/V store.

Return type KeysView

values() → collections.abc.ValuesView

Return view of values in the K/V store.

Return type ValuesView

items() → collections.abc.ItemsView

Return view of items in the K/V store as (key, value) pairs.

Return type ItemsView

clear() → None

Clear all data from this K/V store.

Return type None

logger = <Logger faust.stores.base (WARNING)>

faust.stores.memory

In-memory table storage.

class faust.stores.memory.**Store**(*url: Union[str, yarl.URL], app: faust.types.app.AppT, table: faust.types.tables.CollectionT, *, table_name: str = '', key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, options: Mapping[str, Any] = None, **kwargs: Any*) → None

Table storage using an in-memory dictionary.

apply_changelog_batch(*batch: Iterable[faust.types.events.EventT], to_key: Callable[Any, Any], to_value: Callable[Any, Any]*) → None

Apply batch of changelog events to in-memory table.

Return type None

persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]
Return the persisted offset.

This always returns None when using the in-memory store.

Return type Optional[int]

reset_state() → None
Remove local file system state.

This does nothing when using the in-memory store.

Return type None

logger = <Logger faust.stores.memory (WARNING)>

faust.stores.rocksdb

RocksDB storage.

class faust.stores.rocksdb.**DB**
Dummy DB.

class faust.stores.rocksdb.**Options**
Dummy Options.

class faust.stores.rocksdb.**PartitionDB**(**args*, ***kwargs*)
Tuple of (partition, rocksdb.DB).

property partition
Alias for field number 0

property db
Alias for field number 1

class faust.stores.rocksdb.**RocksDBOptions**(*max_open_files: int = None*, *write_buffer_size: int = None*, *max_write_buffer_number: int = None*, *target_file_size_base: int = None*, *block_cache_size: int = None*, *block_cache_compressed_size: int = None*, *bloom_filter_size: int = None*, ***kwargs: Any*) → None
Options required to open a RocksDB database.

max_open_files = 943719

write_buffer_size = 67108864

max_write_buffer_number = 3

target_file_size_base = 67108864

block_cache_size = 2147483648

block_cache_compressed_size = 524288000

bloom_filter_size = 3

open(*path: pathlib.Path*, ***, *read_only: bool = False*) → faust.stores.rocksdb.DB
Open RocksDB database using this configuration.

Return type *DB*

as_options() → faust.stores.rocksdb.Options
Return `rocksdb.Options` object using this configuration.

Return type *Options*

class faust.stores.rocksdb.**Store**(*url: Union[str, yarl.URL], app: faust.types.app.AppT, table: faust.types.tables.CollectionT, *, key_index_size: int = None, options: Mapping[str, Any] = None, **kwargs: Any*) → None
RocksDB table storage.

offset_key = b'__faust\x00offset__'

rocksdb_options = None
Used to configure the RocksDB settings for table stores.

key_index_size = None
Decides the size of the K=>TopicPartition index (10_000).

persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]
Return the last persisted offset.

See *set_persisted_offset()*.

Return type Optional[int]

set_persisted_offset(*tp: faust.types.tuples.TP, offset: int*) → None
Set the last persisted offset for this table.

This will remember the last offset that we wrote to RocksDB, so that on rebalance/recovery we can seek past this point to only read the events that occurred recently while we were not an active replica.

Return type None

async need_active_standby_for(*tp: faust.types.tuples.TP*) → bool
Decide if an active standby is needed for this topic partition.

Since other workers may be running on the same local machine, we can decide to not actively read standby messages, since that database file is already being populated.

Currently it is recommended that you use separate data directories for multiple worker son the same machine.

For example if you have a 4 CPU core machine, you can run four worker instances on that machine, but using separate data directories:

```
$ myproj --datadir=/var/faust/w1 worker -l info --web-port=6066
$ myproj --datadir=/var/faust/w2 worker -l info --web-port=6067
$ myproj --datadir=/var/faust/w3 worker -l info --web-port=6068
$ myproj --datadir=/var/faust/w4 worker -l info --web-port=6069
```

Return type bool

apply_changelog_batch(*batch: Iterable[faust.types.events.EventT], to_key: Callable[Any, Any], to_value: Callable[Any, Any]*) → None
Write batch of changelog events to local RocksDB storage.

Parameters

- **batch** (Iterable[*EventT*[]]) – Iterable of changelog events (*faust.Event*)
- **to_key** (Callable[[Any], Any]) – A callable you can use to deserialize the key of a changelog event.
- **to_value** (Callable[[Any], Any]) – A callable you can use to deserialize the value of a changelog event.

Return type None

async on_rebalance(*table: faust.types.tables.CollectionT, assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None

Rebalance occurred.

Parameters

- **table** (*CollectionT*[]) – The table that we store data for.
- **assigned** (Set[*TP*]) – Set of all assigned topic partitions.
- **revoked** (Set[*TP*]) – Set of newly revoked topic partitions.
- **newly_assigned** (Set[*TP*]) – Set of newly assigned topic partitions, for which we were not assigned the last time.

Return type None

revoke_partitions(*table: faust.types.tables.CollectionT, tps: Set[faust.types.tuples.TP]*) → None

De-assign partitions used on this worker instance.

Parameters

- **table** (*CollectionT*[]) – The table that we store data for.
- **tps** (Set[*TP*]) – Set of topic partitions that we should no longer be serving data for.

Return type None

async assign_partitions(*table: faust.types.tables.CollectionT, tps: Set[faust.types.tuples.TP]*) → None

Assign partitions to this worker instance.

Parameters

- **table** (*CollectionT*[]) – The table that we store data for.
- **tps** (Set[*TP*]) – Set of topic partitions we have been assigned.

Return type None

logger = <Logger faust.stores.rocksdb (WARNING)>

reset_state() → None

Remove all data stored in this table.

Notes

Only local data will be removed, table changelog partitions in Kafka will not be affected.

Return type None

partition_path(*partition: int*) → pathlib.Path

Return pathlib.Path to db file of specific partition.

Return type Path

property path

Path to directory where tables are stored.

See also:

tabledir (default value for this path).

Return type Path

Returns pathlib.Path.

property basename
Return the name of this table, used as filename prefix. :rtype: Path

1.6.10 Tables

faust.tables

Tables: Distributed object K/V-store.

class faust.tables.**Collection**(*app: faust.types.app.AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, recover_callbacks: Set[Callable[Awaitable[None]]] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None
Base class for changelog-backed data structures stored in Kafka.

property data
Underlying table storage.

async on_start() → None
Call when table starts.

Return type None

on_recover(*fun: Callable[Awaitable[None]]*) → Callable[Awaitable[None]]
Add function as callback to be called on table recovery.

Return type Callable[[], Awaitable[None]]

info() → Mapping[str, Any]
Return table attributes as dictionary.

Return type Mapping[str, Any]

persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]
Return the last persisted offset for topic partition.

Return type Optional[int]

async need_active_standby_for(*tp: faust.types.tuples.TP*) → bool
Return False if we have access to partition data.

Return type bool

reset_state() → None
Reset local state.

Return type None

send_changelog(*partition: Optional[int], key: Any, value: Any, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → faust.types.tuples.FutureMessage
Send modification event to changelog topic.

Return type *FutureMessage*[]

partition_for_key(*key: Any*) → Optional[int]
Return partition number for table key. c

Note: If `use_partitioner` is enabled this always returns `None`.

Returns:

Optional[int]: specific partition or `None` if the producer should select partition using its partitioner.

Return type Optional[int]

on_window_close(*key: Any*, *value: Any*) → None

Return type None

join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Right join of this table and another stream/table.

Return type *StreamT*[+T_co]

left_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Left join of this table and another stream/table.

Return type *StreamT*[+T_co]

inner_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Inner join of this table and another stream/table.

Return type *StreamT*[+T_co]

outer_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Outer join of this table and another stream/table.

Return type *StreamT*[+T_co]

clone(***kwargs: Any*) → Any
Clone table instance.

Return type Any

combine(**nodes: faust.types.streams.JoinableT*, ***kwargs: Any*) → faust.types.streams.StreamT
Combine tables and streams.

Return type *StreamT*[+T_co]

contribute_to_stream(*active: faust.types.streams.StreamT*) → None
Contribute table to stream join.

Return type None

async remove_from_stream(*stream: faust.types.streams.StreamT*) → None
Remove table from stream join after stream stopped.

Return type None

async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing.

Return type None

async on_recovery_completed(*active_tps: Set[faust.types.tuples.TP], standby_tps: Set[faust.types.tuples.TP]*) → None
Call when recovery has completed after rebalancing.

Return type None

async call_recover_callbacks() → None
Call any configured recovery callbacks after rebalancing.

Return type None

async on_changelog_event(*event: faust.types.events.EventT*) → None
Call when a new changelog event is received.

Return type None

property label
Return human-readable label used to represent this table. :rtype: str

property shortlabel
Return short label used to represent this table in logs. :rtype: str

property changelog_topic
Return the changelog topic used by this table. :rtype: *TopicT*[]

logger = <Logger faust.tables.base (WARNING)>

property changelog_topic_name

Return type str

apply_changelog_batch(*batch: Iterable[faust.types.events.EventT]*) → None
Apply batch of events from changelog topic local table storage.

Return type None

class faust.tables.**CollectionT**(*app: faust.types.tables._AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.tables._SchemaT = None, key_type: faust.types.tables._ModelArg = None, value_type: faust.types.tables._ModelArg = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

abstract property changelog_topic

Return type *TopicT*[]

abstract apply_changelog_batch(*batch: Iterable[faust.types.events.EventT]*) → None

Return type None

abstract persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]

Return type Optional[int]

abstract async need_active_standby_for(*tp: faust.types.tuples.TP*) → bool

Return type bool

abstract reset_state() → None

Return type None

abstract send_changelog(*partition: Optional[int], key: Any, value: Any, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → faust.types.tuples.FutureMessage

Return type *FutureMessage*[]

abstract partition_for_key(*key: Any*) → Optional[int]

Return type Optional[int]

abstract on_window_close(*key: Any*, *value: Any*) → None

Return type None

abstract async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async on_changelog_event(*event: faust.types.events.EventT*) → None

Return type None

abstract on_recover(*fun: Callable[Awaitable[None]]*) → Callable[Awaitable[None]]

Return type Callable[[], Awaitable[None]]

abstract async on_recovery_completed(*active_tps: Set[faust.types.tuples.TP], standby_tps: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async call_recover_callbacks() → None

Return type None

class faust.tables.**TableManager**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None

Manage tables used by Faust worker.

persist_offset_on_commit(*store: faust.types.stores.StoreT*, *tp: faust.types.tuples.TP*, *offset: int*) → None

Mark the persisted offset for a TP to be saved on commit.

This is used for "exactly_once" processing guarantee. Instead of writing the persisted offset to RocksDB when the message is sent, we write it to disk when the offset is committed.

Return type None

on_commit(*offsets: MutableMapping[faust.types.tuples.TP, int]*) → None

Call when committing source topic partitions.

Return type None

on_commit_tp(*tp: faust.types.tuples.TP*) → None

Call when committing source topic partition used by this table.

Return type None

on_rebalance_start() → None
Call when a new rebalancing operation starts.

Return type None

on_actives_ready() → None
Call when actives are fully up-to-date.

Return type None

on_standbys_ready() → None
Call when standbys are fully up-to-date and ready for failover.

Return type None

property changelog_topics
Return the set of known changelog topics. :rtype: Set[str]

property changelog_queue
Queue used to buffer changelog events. :rtype: ThrowableQueue

property recovery
Recovery service used by this table manager. :rtype: *Recovery*[]

add(*table: faust.types.tables.CollectionT*) → faust.types.tables.CollectionT
Add table to be managed by this table manager.

Return type *CollectionT*[]

logger = <Logger faust.tables.manager (WARNING)>

async on_start() → None
Call when table manager is starting.

Return type None

async on_stop() → None
Call when table manager is stopping.

Return type None

on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing and partitions revoked.

Return type None

async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None
Call when the cluster is rebalancing.

Return type None

class faust.tables.**TableManagerT**(*app: faust.types.tables._AppT*, ***kwargs: Any*) → None

abstract add(*table: faust.types.tables.CollectionT*) → faust.types.tables.CollectionT

Return type *CollectionT*[]

abstract persist_offset_on_commit(*store: faust.types.stores.StoreT*, *tp: faust.types.tuples.TP*, *offset: int*) → None

Return type None

abstract on_commit(*offsets: MutableMapping[faust.types.tuples.TP, int]*) → None

Return type None

abstract async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None

Return type None

abstract property changelog_topics

Return type Set[str]

class faust.tables.**Table**(*app: faust.types.app.AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, recover_callbacks: Set[Callable[Awaitable[None]]] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

Table (non-windowed).

class WindowWrapper(*table: faust.types.tables.TableT, *, relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None] = None, key_index: bool = False, key_index_table: faust.types.tables.TableT = None*) → None

Windowed table wrapper.

A windowed table does not return concrete values when keys are accessed, instead WindowSet is returned so that the values can be further reduced to the wanted time period.

ValueType

alias of WindowSet

as_ansitable(*title: str = '{table.name}', **kwargs: Any*) → str

Draw table as a terminal ANSI table.

Return type str

clone(*relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT

Clone this table using a new time-relativity configuration.

Return type *WindowWrapperT*[]

property get_relative_timestamp

Return the current handler for extracting event timestamp. :rtype: Optional[Callable[[Optional[*EventT*[]]], Union[float, datetime]]]

get_timestamp(*event: faust.types.events.EventT = None*) → float

Get timestamp from event.

Return type float

items(*event: faust.types.events.EventT = None*) → ItemsView
Return table items view: iterate over `(key, value)` pairs.
Return type `ItemsView`[~KT, +VT_co]

key_index = False

key_index_table = None

keys() → KeysView
Return table keys view: iterate over keys found in this table.
Return type `KeysView`[~KT]

property name
Return the name of this table. :rtype: `str`

on_del_key(*key: Any*) → None
Call when a key is deleted from this table.
Return type `None`

on_recover(*fun: Callable[Awaitable[None]]*) → Callable[Awaitable[None]]
Call after table recovery.
Return type `Callable`[[], `Awaitable`[None]]

on_set_key(*key: Any*, *value: Any*) → None
Call when the value for a key in this table is set.
Return type `None`

relative_to(*ts: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT
Configure the time-relativity of this windowed table.
Return type *WindowWrapperT*[]

relative_to_field(*field: faust.types.models.FieldDescriptorT*) → faust.types.tables.WindowWrapperT
Configure table to be time-relative to a field in the stream.

This means the window will use the timestamp from the event currently being processed in the stream.

Further it will not use the timestamp of the Kafka message, but a field in the value of the event.

For example a model field:

```
class Account(faust.Record):
    created: float

table = app.Table('foo').hopping(
    ...,
).relative_to_field(Account.created)
```

Return type *WindowWrapperT*[]

relative_to_now() → faust.types.tables.WindowWrapperT
Configure table to be time-relative to the system clock.
Return type *WindowWrapperT*[]

relative_to_stream() → faust.types.tables.WindowWrapperT
Configure table to be time-relative to the stream.

This means the window will use the timestamp from the event currently being processed in the stream.
Return type *WindowWrapperT*[]

values(*event: faust.types.events.EventT = None*) → ValuesView
Return table values view: iterate over values in this table.
Return type ValuesView[+VT_co]

using_window(*window: faust.types.windows.WindowT*, *, *key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table using a specific window type.

Return type *WindowWrapperT*[]

hopping(*size: Union[datetime.timedelta, float, str], step: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table in a hopping window.

Return type *WindowWrapperT*[]

tumbling(*size: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table in a tumbling window.

Return type *WindowWrapperT*[]

on_key_get(*key: KT*) → None
Call when the value for a key in this table is retrieved.

Return type None

on_key_set(*key: KT*, *value: VT*) → None
Call when the value for a key in this table is set.

Return type None

on_key_del(*key: KT*) → None
Call when a key in this table is removed.

Return type None

as_ansitable(*title: str = '{table.name}'*, ***kwargs: Any*) → str
Draw table as a a terminal ANSI table.

Return type str

logger = <Logger faust.tables.table (WARNING)>

class faust.tables.**TableT**(*app: faust.types.tables._AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.tables._SchemaT = None, key_type: faust.types.tables._ModelArg = None, value_type: faust.types.tables._ModelArg = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

abstract using_window(*window: faust.types.windows.WindowT*, *, *key_index: bool = False*) → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract hopping(*size: Union[datetime.timedelta, float, str], step: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract tumbling(*size: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract as_ansitable(***kwargs: Any*) → str

Return type str

faust.tables.base

Base class Collection for Table and future data structures.

class faust.tables.base.**Collection**(*app: faust.types.app.AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, recover_callbacks: Set[Callable[Awaitable[None]]] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

Base class for changelog-backed data structures stored in Kafka.

property data
Underlying table storage.

async on_start() → None
Call when table starts.

Return type None

on_recover(*fun: Callable[Awaitable[None]]*) → Callable[Awaitable[None]]
Add function as callback to be called on table recovery.

Return type Callable[[], Awaitable[None]]

info() → Mapping[str, Any]
Return table attributes as dictionary.

Return type Mapping[str, Any]

persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]
Return the last persisted offset for topic partition.

Return type Optional[int]

async need_active_standby_for(*tp: faust.types.tuples.TP*) → bool
Return False if we have access to partition data.

Return type bool

reset_state() → None
Reset local state.

Return type None

send_changelog(*partition: Optional[int], key: Any, value: Any, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → faust.types.tuples.FutureMessage
Send modification event to changelog topic.

Return type *FutureMessage*[]

partition_for_key(*key: Any*) → Optional[int]
Return partition number for table key. c

Note: If use_partitioner is enabled this always returns None.

Returns:

Optional[int]: specific partition or None if the producer should select partition using its partitioner.

Return type Optional[int]

on_window_close(*key: Any*, *value: Any*) → None

Return type None

join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Right join of this table and another stream/table.

Return type *StreamT*[+T_co]

left_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Left join of this table and another stream/table.

Return type *StreamT*[+T_co]

inner_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Inner join of this table and another stream/table.

Return type *StreamT*[+T_co]

outer_join(**fields: faust.types.models.FieldDescriptorT*) → faust.types.streams.StreamT
Outer join of this table and another stream/table.

Return type *StreamT*[+T_co]

clone(***kwargs: Any*) → Any
Clone table instance.

Return type Any

combine(**nodes: faust.types.streams.JoinableT*, ***kwargs: Any*) → faust.types.streams.StreamT
Combine tables and streams.

Return type *StreamT*[+T_co]

contribute_to_stream(*active: faust.types.streams.StreamT*) → None
Contribute table to stream join.

Return type None

async remove_from_stream(*stream: faust.types.streams.StreamT*) → None
Remove table from stream join after stream stopped.

Return type None

async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing.

Return type None

async on_recovery_completed(*active_tps: Set[faust.types.tuples.TP], standby_tps: Set[faust.types.tuples.TP]*) → None
Call when recovery has completed after rebalancing.

Return type None

async call_recover_callbacks() → None
Call any configured recovery callbacks after rebalancing.

Return type None

async on_changelog_event(*event: faust.types.events.EventT*) → None
Call when a new changelog event is received.

Return type None

property label
Return human-readable label used to represent this table. :rtype: str

property shortlabel
Return short label used to represent this table in logs. :rtype: str

property changelog_topic
Return the changelog topic used by this table. :rtype: *TopicT*[]

logger = <Logger faust.tables.base (WARNING)>

property changelog_topic_name

Return type str

apply_changelog_batch(*batch: Iterable[faust.types.events.EventT]*) → None
Apply batch of events from changelog topic local table storage.

Return type None

faust.tables.globaltable

class faust.tables.globaltable.**GlobalTable**(*app: faust.types.app.AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, recover_callbacks: Set[Callable[Awaitable[None]]] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

logger = <Logger faust.tables.globaltable (WARNING)>

faust.tables.manager

Tables (changelog stream).

class faust.tables.manager.**TableManager**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None

Manage tables used by Faust worker.

persist_offset_on_commit(*store: faust.types.stores.StoreT*, *tp: faust.types.tuples.TP*, *offset: int*) → None

Mark the persisted offset for a TP to be saved on commit.

This is used for "exactly_once" processing guarantee. Instead of writing the persisted offset to RocksDB when the message is sent, we write it to disk when the offset is committed.

Return type None

on_commit(*offsets: MutableMapping[faust.types.tuples.TP, int]*) → None

Call when committing source topic partitions.

Return type None

on_commit_tp(*tp: faust.types.tuples.TP*) → None

Call when committing source topic partition used by this table.

Return type None

on_rebalance_start() → None

Call when a new rebalancing operation starts.

Return type None

on_actives_ready() → None
Call when actives are fully up-to-date.

Return type None

on_standbys_ready() → None
Call when standbys are fully up-to-date and ready for failover.

Return type None

property changelog_topics
Return the set of known changelog topics. :rtype: Set[str]

property changelog_queue
Queue used to buffer changelog events. :rtype: ThrowableQueue

property recovery
Recovery service used by this table manager. :rtype: *Recovery*[]

add(*table: faust.types.tables.CollectionT*) → faust.types.tables.CollectionT
Add table to be managed by this table manager.

Return type *CollectionT*[]

logger = <Logger faust.tables.manager (WARNING)>

async on_start() → None
Call when table manager is starting.

Return type None

async on_stop() → None
Call when table manager is stopping.

Return type None

on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing and partitions revoked.

Return type None

async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None
Call when the cluster is rebalancing.

Return type None

faust.tables.objects

Storing objects in tables.

This is also used to store data structures such as sets/lists.

class faust.tables.objects.**ChangeloggedObject**(*manager: faust.tables.objects.ChangeloggedObjectManager*, *key: Any*) → None
A changelogged object in a *ChangeloggedObjectManager* store.

abstract sync_from_storage(*value: Any*) → None
Sync value from storage.

Return type None

abstract as_stored_value() → Any
Return value as represented in storage.

Return type Any

abstract apply_changelog_event(*operation: int*, *value: Any*) → None
Apply event in changelog topic to local table state.

Return type None

class faust.tables.objects.**ChangeloggedObjectManager**(*table: faust.tables.table.Table*, ***kwargs: Any*) → None
Store of changelogged objects.

send_changelog_event(*key: Any*, *operation: int*, *value: Any*) → None
Send changelog event to the tables changelog topic.

Return type None

async on_start() → None
Call when the changelogged object manager starts.

Return type None

async on_stop() → None
Call when the changelogged object manager stops.

Return type None

persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]
Get the last persisted offset for changelog topic partition.

Return type Optional[int]

set_persisted_offset(*tp: faust.types.tuples.TP*, *offset: int*) → None
Set the last persisted offset for changelog topic partition.

Return type None

async on_rebalance(*table: faust.types.tables.CollectionT*, *assigned: Set[faust.types.tuples.TP]*, *revoked: Set[faust.types.tuples.TP]*, *newly_assigned: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing.

Return type None

async on_recovery_completed(*active_tps: Set[faust.types.tuples.TP]*, *standby_tps: Set[faust.types.tuples.TP]*) → None
Call when table recovery is completed after rebalancing.

Return type None

sync_from_storage() → None
Sync set contents from storage.

Return type None

flush_to_storage() → None
Flush set contents to storage.

Return type None

logger = <Logger faust.tables.objects (WARNING)>

reset_state() → None
Reset table local state.

Return type None

property storage
Return underlying storage used by this set table. :rtype: *StoreT*[~KT, ~VT]

apply_changelog_batch(*batch: Iterable[faust.types.events.EventT], to_key: Callable[Any, Any], to_value: Callable[Any, Any]*) → None
Apply batch of changelog events to local state.

Return type None

faust.tables.recovery

Table recovery after rebalancing.

exception faust.tables.recovery.**ServiceStopped**
The recovery service was stopped.

exception faust.tables.recovery.**RebalanceAgain**
During rebalance, another rebalance happened.

class faust.tables.recovery.**Recovery**(*app: faust.types.app.AppT, tables: faust.types.tables.TableManagerT, **kwargs: Any*) → None
Service responsible for recovering tables from changelog topics.

stats_interval = 5.0

highwaters = None
Mapping of highwaters by topic partition.

in_recovery = False

standbys_pending = False

standby_tps = None
Set of standby topic partitions.

active_tps = None
Set of active topic partitions.

tp_to_table = None
Mapping from topic partition to table

active_offsets = None
Active offset by topic partition.

standby_offsets = None
Standby offset by topic partition.

active_highwaters = None
Active highwaters by topic partition.

standby_highwaters = None
Standby highwaters by topic partition.

buffers = None
Changelog event buffers by table. These are filled by background task *_slurp_changelog*, and need to be flushed before starting new recovery/stopping.

buffer_sizes = None
Cache of buffer size by topic partition..

property signal_recovery_start
Event used to signal that recovery has started. :rtype: `Event`

property signal_recovery_end
Event used to signal that recovery has ended. :rtype: `Event`

property signal_recovery_reset
Event used to signal that recovery is restarting. :rtype: `Event`

async on_stop() → None
Call when recovery service stops.

Return type `None`

add_active(*table: faust.types.tables.CollectionT*, *tp: faust.types.tuples.TP*) → None
Add changelog partition to be used for active recovery.

Return type `None`

add_standby(*table: faust.types.tables.CollectionT*, *tp: faust.types.tuples.TP*) → None
Add changelog partition to be used for standby recovery.

Return type `None`

add_standbys_for_global_table(*gtable: faust.types.tables.CollectionT*, *active_topic: faust.types.tuples.TP*, *topics: Set[faust.types.tuples.TP]*) → None
Add standby topics for global table.

Return type `None`

revoke(*tp: faust.types.tuples.TP*) → None
Revoke assignment of table changelog partition.

Return type `None`

on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when rebalancing and partitions are revoked.

Return type `None`

async on_rebalance(*assigned: Set[faust.types.tuples.TP]*, *revoked: Set[faust.types.tuples.TP]*, *newly_assigned: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing.

Return type `None`

logger = <Logger faust.tables.recovery (WARNING)>

async on_recovery_completed() → None
Call when active table recovery is completed.

Return type `None`

flush_buffers() → None
Flush changelog buffers.

Return type `None`

need_recovery() → bool
Return `True` if recovery is required.

Return type `bool`

active_remaining() → Counter[faust.types.tuples.TP]
Return counter of remaining changes by active partition.

Return type Counter[*TP*]

standby_remaining() → Counter[faust.types.tuples.TP]
Return counter of remaining changes by standby partition.

Return type Counter[*TP*]

active_remaining_total() → int
Return number of changes remaining for actives to be up-to-date.

Return type int

standby_remaining_total() → int
Return number of changes remaining for standbys to be up-to-date.

Return type int

active_stats() → MutableMapping[faust.types.tuples.TP, Tuple[int, int, int]]
Return current active recovery statistics.

Return type MutableMapping[*TP*, Tuple[int, int, int]]

standby_stats() → MutableMapping[faust.types.tuples.TP, Tuple[int, int, int]]
Return current standby recovery statistics.

Return type MutableMapping[*TP*, Tuple[int, int, int]]

faust.tables.sets

Storing sets in tables.

class faust.tables.sets.**SetTable**(*app: faust.types.app.AppT*, ***, *start_manager: bool = False*, *manager_topic_name: str = None*, *manager_topic_suffix: str = None*, ***kwargs: Any*) → None
Table that maintains a dictionary of sets.

Manager
alias of SetTableManager

WindowWrapper
alias of SetWindowWrapper

logger = <Logger faust.tables.sets (WARNING)>

manager_topic_suffix = '-setmanager'

async on_start() → None
Call when set table starts.

Return type None

class faust.tables.sets.**SetGlobalTable**(*app: faust.types.app.AppT*, ***, *start_manager: bool = False*, *manager_topic_name: str = None*, *manager_topic_suffix: str = None*, ***kwargs: Any*) → None

logger = <Logger faust.tables.sets (WARNING)>

faust.tables.table

Table (key/value changelog stream).

class faust.tables.table.**Table**(*app: faust.types.app.AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, recover_callbacks: Set[Callable[Awaitable[None]]] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

Table (non-windowed).

class WindowWrapper(*table: faust.types.tables.TableT, *, relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None] = None, key_index: bool = False, key_index_table: faust.types.tables.TableT = None*) → None

Windowed table wrapper.

A windowed table does not return concrete values when keys are accessed, instead `WindowSet` is returned so that the values can be further reduced to the wanted time period.

ValueType
alias of `WindowSet`

as_ansitable(*title: str = '{table.name}', **kwargs: Any*) → str
Draw table as a terminal ANSI table.
Return type `str`

clone(*relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT
Clone this table using a new time-relativity configuration.
Return type *WindowWrapperT*[]

property get_relative_timestamp
Return the current handler for extracting event timestamp. :rtype: `Optional`[`Callable`[[`Optional`[*EventT*[]]], `Union`[`float`, `datetime`]]]

get_timestamp(*event: faust.types.events.EventT = None*) → float
Get timestamp from event.
Return type `float`

items(*event: faust.types.events.EventT = None*) → ItemsView
Return table items view: iterate over `(key, value)` pairs.
Return type `ItemsView`[~KT, +VT_co]

key_index = False

key_index_table = None

keys() → KeysView
Return table keys view: iterate over keys found in this table.
Return type KeysView[~KT]

property name
Return the name of this table. :rtype: str

on_del_key(*key: Any*) → None
Call when a key is deleted from this table.
Return type None

on_recover(*fun: Callable[Awaitable[None]]*) → Callable[Awaitable[None]]
Call after table recovery.
Return type Callable[[], Awaitable[None]]

on_set_key(*key: Any*, *value: Any*) → None
Call when the value for a key in this table is set.
Return type None

relative_to(*ts: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT
Configure the time-relativity of this windowed table.
Return type *WindowWrapperT*[]

relative_to_field(*field: faust.types.models.FieldDescriptorT*) → faust.types.tables.WindowWrapperT
Configure table to be time-relative to a field in the stream.

This means the window will use the timestamp from the event currently being processed in the stream.

Further it will not use the timestamp of the Kafka message, but a field in the value of the event.

For example a model field:

```
class Account(faust.Record):
    created: float

table = app.Table('foo').hopping(
    ...,
).relative_to_field(Account.created)
```

Return type *WindowWrapperT*[]

relative_to_now() → faust.types.tables.WindowWrapperT
Configure table to be time-relative to the system clock.
Return type *WindowWrapperT*[]

relative_to_stream() → faust.types.tables.WindowWrapperT
Configure table to be time-relative to the stream.

This means the window will use the timestamp from the event currently being processed in the stream.
Return type *WindowWrapperT*[]

values(*event: faust.types.events.EventT = None*) → ValuesView
Return table values view: iterate over values in this table.
Return type ValuesView[+VT_co]

using_window(*window: faust.types.windows.WindowT*, ***, *key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table using a specific window type.

Return type *WindowWrapperT*[]

hopping(*size: Union[datetime.timedelta, float, str], step: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table in a hopping window.

Return type *WindowWrapperT*[]

tumbling(*size: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT
Wrap table in a tumbling window.

Return type *WindowWrapperT*[]

on_key_get(*key: KT*) → None
Call when the value for a key in this table is retrieved.

Return type None

on_key_set(*key: KT*, *value: VT*) → None
Call when the value for a key in this table is set.

Return type None

on_key_del(*key: KT*) → None
Call when a key in this table is removed.

Return type None

as_ansitable(*title: str = '{table.name}'*, ***kwargs: Any*) → str
Draw table as a a terminal ANSI table.

Return type str

logger = <Logger faust.tables.table (WARNING)>

faust.tables.wrappers

Wrappers for windowed tables.

class faust.tables.wrappers.**WindowedKeysView**(*mapping: faust.types.tables.WindowWrapperT*, *event: faust.types.events.EventT = None*)
The object returned by `windowed_table.keys()`.

now() → Iterator[Any]
Return all keys present in window closest to system time.

Return type Iterator[Any]

current(*event: faust.types.events.EventT = None*) → Iterator[Any]
Return all keys present in window closest to stream time.

Return type Iterator[Any]

delta(*d: Union[datetime.timedelta, float, str], event: faust.types.events.EventT = None*) → Iterator[Any]
Return all keys present in window ±n seconds ago.

Return type Iterator[Any]

class faust.tables.wrappers.**WindowedItemsView**(*mapping: faust.types.tables.WindowWrapperT*, *event: faust.types.events.EventT = None*)
The object returned by `windowed_table.items()`.

now() → Iterator[Tuple[Any, Any]]
Return all items present in window closest to system time.

Return type Iterator[Tuple[Any, Any]]

current(*event: faust.types.events.EventT = None*) → Iterator[Tuple[Any, Any]]
Return all items present in window closest to stream time.

Return type Iterator[Tuple[Any, Any]]

delta(*d: Union[datetime.timedelta, float, str], event: faust.types.events.EventT = None*) → Iterator[Tuple[Any, Any]]
Return all items present in window ±n seconds ago.

Return type Iterator[Tuple[Any, Any]]

class faust.tables.wrappers.**WindowedValuesView**(*mapping: faust.types.tables.WindowWrapperT, event: faust.types.events.EventT = None*)
The object returned by `windowed_table.values()`.

now() → Iterator[Any]
Return all values present in window closest to system time.

Return type Iterator[Any]

current(*event: faust.types.events.EventT = None*) → Iterator[Any]
Return all values present in window closest to stream time.

Return type Iterator[Any]

delta(*d: Union[datetime.timedelta, float, str], event: faust.types.events.EventT = None*) → Iterator[Any]
Return all values present in window ±n seconds ago.

Return type Iterator[Any]

class faust.tables.wrappers.**WindowSet**(*key: KT, table: faust.types.tables.TableT, wrapper: faust.types.tables.WindowWrapperT, event: faust.types.events.EventT = None*) → None
Represents the windows available for table key.

`Table[k]` returns WindowSet since `k` can exist in multiple windows, and to retrieve an actual item we need a timestamp.

The timestamp of the current event (if this is executing in a stream processor), can be used by accessing `.current()`:

```
Table[k].current()
```

similarly the most recent value can be accessed using `.now()`:

```
Table[k].now()
```

from delta of the time of the current event:

```
Table[k].delta(timedelta(hours=3))
```

or delta from time of other event:

```
Table[k].delta(timedelta(hours=3), other_event)
```

apply(*op: Callable[[VT, VT], VT], value: VT, event: faust.types.events.EventT = None*) → faust.types.tables.WindowSetT[KT, VT]
Apply operation to all affected windows.

Return type *WindowSetT*[~KT, ~VT]

value(*event: faust.types.events.EventT = None*) → VT
Return current value.

The selected window depends on the current time-relativity setting used (`relative_to_now()`, `relative_to_stream()`, `relative_to_field()`, etc.)

Return type ~VT

now() → VT
Return current value, using the current system time.

Return type ~VT

current(*event: faust.types.events.EventT = None*) → VT
Return current value, using stream time-relativity.

Return type ~VT

delta(*d: Union[datetime.timedelta, float, str], event: faust.types.events.EventT = None*) → VT
Return value as it was ±n seconds ago.

Return type ~VT

class faust.tables.wrappers.**WindowWrapper**(*table: faust.types.tables.TableT, *, relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None] = None, key_index: bool = False, key_index_table: faust.types.tables.TableT = None*) → None
Windowed table wrapper.

A windowed table does not return concrete values when keys are accessed, instead *WindowSet* is returned so that the values can be further reduced to the wanted time period.

ValueType
alias of *WindowSet*

key_index = False

key_index_table = None

clone(*relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT
Clone this table using a new time-relativity configuration.

Return type *WindowWrapperT*[]

property name
Return the name of this table. :rtype: `str`

relative_to(*ts: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT
Configure the time-relativity of this windowed table.

Return type *WindowWrapperT*[]

relative_to_now() → faust.types.tables.WindowWrapperT
Configure table to be time-relative to the system clock.

Return type *WindowWrapperT*[]

relative_to_field(*field: faust.types.models.FieldDescriptorT*) → faust.types.tables.WindowWrapperT

Configure table to be time-relative to a field in the stream.

This means the window will use the timestamp from the event currently being processed in the stream.

Further it will not use the timestamp of the Kafka message, but a field in the value of the event.

For example a model field:

```
class Account(faust.Record):
    created: float

table = app.Table('foo').hopping(
    ...,
).relative_to_field(Account.created)
```

Return type *WindowWrapperT*[]

relative_to_stream() → faust.types.tables.WindowWrapperT

Configure table to be time-relative to the stream.

This means the window will use the timestamp from the event currently being processed in the stream.

Return type *WindowWrapperT*[]

get_timestamp(*event: faust.types.events.EventT = None*) → float

Get timestamp from event.

Return type `float`

on_recover(*fun: Callable[Awaitable[None]]*) → Callable[Awaitable[None]]

Call after table recovery.

Return type `Callable`[[], `Awaitable`[`None`]]

on_set_key(*key: Any*, *value: Any*) → None

Call when the value for a key in this table is set.

Return type `None`

on_del_key(*key: Any*) → None

Call when a key is deleted from this table.

Return type `None`

keys() → KeysView

Return table keys view: iterate over keys found in this table.

Return type `KeysView`[~KT]

values(*event: faust.types.events.EventT = None*) → ValuesView

Return table values view: iterate over values in this table.

Return type `ValuesView`[+VT_co]

items(*event: faust.types.events.EventT = None*) → ItemsView

Return table items view: iterate over `(key, value)` pairs.

Return type `ItemsView`[~KT, +VT_co]

as_ansitable(*title: str = '{table.name}'*, ***kwargs: Any*) → str

Draw table as a terminal ANSI table.

Return type `str`

property get_relative_timestamp
Return the current handler for extracting event timestamp. :rtype: Optional[Callable[[Optional[*EventT*[]]], Union[float, datetime]]]

1.6.11 Transports

faust.transport

Transports.

faust.transport.**by_name**(*name: Union[_T, str]*) → _T

Return type ~_T

faust.transport.**by_url**(*url: Union[str, yarl.URL]*) → _T
Get class associated with URL (scheme is used as alias key).

Return type ~_T

faust.transport.base

Base message transport implementation.

The Transport is responsible for:

- Holds reference to the app that created it.
- Creates new consumers/producers.

To see a reference transport implementation go to: `faust/transport/drivers/aiokafka.py`

class faust.transport.base.**Conductor**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Manages the channels that subscribe to topics.

- Consumes messages from topic using a single consumer.
- Forwards messages to all channels subscribing to a topic.

logger = <Logger faust.transport.conductor (WARNING)>

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]]*) → bool
Commit offsets in topics.

Return type bool

acks_enabled_for(*topic: str*) → bool
Return `True` if acks are enabled for topic by name.

Return type bool

async wait_for_subscriptions() → None
Wait for consumer to be subscribed.

Return type None

async maybe_wait_for_subscriptions() → None

Return type None

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing and partitions are assigned.

Return type None

async on_client_only_start() → None

Return type None

clear() → None
Clear all subscriptions.

Return type None

add(*topic: Any*) → None
Register topic to be subscribed.

Return type None

discard(*topic: Any*) → None
Unregister topic from conductor.

Return type None

property label
Return label for use in logs. :rtype: str

property shortlabel
Return short label for use in logs. :rtype: str

class faust.transport.base.**Consumer**(*transport: faust.types.transports.TransportT, callback: Callable[faust.types.tuples.Message, Awaitable], on_partitions_revoked: Callable[Set[faust.types.tuples.TP], Awaitable[None]], on_partitions_assigned: Callable[Set[faust.types.tuples.TP], Awaitable[None]], *, commit_interval: float = None, commit_livelock_soft_timeout: float = None, loop: asyncio.events.AbstractEventLoop = None, **kwargs: Any*) → None

Base Consumer.

logger = <Logger faust.transport.consumer (WARNING)>

consumer_stopped_errors = ()
Tuple of exception types that may be raised when the underlying consumer driver is stopped.

flow_active = True

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of services this consumer depends on.

Return type Iterable[ServiceT[]]

async on_restart() → None
Call when the consumer is restarted.

Return type None

async perform_seek() → None
Seek all partitions to their current committed position.

Return type None

abstract async seek_to_committed() → Mapping[faust.types.tuples.TP, int]
Seek all partitions to their committed offsets.

Return type Mapping[*TP*, int]

async seek(*partition: faust.types.tuples.TP*, *offset: int*) → None
Seek partition to specific offset.

Return type None

stop_flow() → None
Block consumer from processing any more messages.

Return type None

resume_flow() → None
Allow consumer to process messages.

Return type None

pause_partitions(*tps: Iterable[faust.types.tuples.TP]*) → None
Pause fetching from partitions.

Return type None

resume_partitions(*tps: Iterable[faust.types.tuples.TP]*) → None
Resume fetching from partitions.

Return type None

async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call during rebalancing when partitions are being revoked.

Return type None

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call during rebalancing when partitions are being assigned.

Return type None

getmany(*timeout: float*) → AsyncIterator[Tuple[faust.types.tuples.TP, faust.types.tuples.Message]]
Fetch batch of messages from server.

Return type AsyncIterator[Tuple[*TP*, *Message*]]

track_message(*message: faust.types.tuples.Message*) → None
Track message and mark it as pending ack.

Return type None

ack(*message: faust.types.tuples.Message*) → bool
Mark message as being acknowledged by stream.

Return type bool

async wait_empty() → None
Wait for all messages that started processing to be acked.

Return type None

async commit_and_end_transactions() → None
Commit all safe offsets and end transaction.

Return type None

async on_stop() → None
Call when consumer is stopping.

Return type None

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]] = None*, *start_new_transaction: bool = True*) → bool
Maybe commit the offset for all or specific topics.

Parameters **topics** (Optional[AbstractSet[Union[str, *TP*]]]) – Set containing topics and/or TopicPartitions to commit.

Return type bool

async maybe_wait_for_commit_to_finish() → bool
Wait for any existing commit operation to finish.

Return type bool

async force_commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]] = None*, *start_new_transaction: bool = True*) → bool
Force offset commit.

Return type bool

async on_task_error(*exc: BaseException*) → None
Call when processing a message failed.

Return type None

close() → None
Close consumer for graceful shutdown.

Return type None

property unacked
Return the set of currently unacknowledged messages. :rtype: Set[*Message*]

class faust.transport.base.**Fetcher**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Service fetching messages from Kafka.

logger = <Logger faust.transport.consumer (WARNING)>

async on_stop() → None
Call when the fetcher is stopping.

Return type None

class faust.transport.base.**Producer**(*transport: faust.types.transports.TransportT*, *loop: asyncio.events.AbstractEventLoop = None*, ***kwargs: Any*) → None
Base Producer.

async on_start() → None
Service is starting.

Return type None

async send(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]
Schedule message to be sent by producer.

Return type Awaitable[*RecordMetadata*]

send_soon(*fut: faust.types.tuples.FutureMessage*) → None

Return type None

async send_and_wait(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → faust.types.tuples.RecordMetadata
Send message and wait for it to be transmitted.

Return type *RecordMetadata*

async flush() → None
Flush all in-flight messages.

Return type None

async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 1000.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None
Create/declare topic on server.

Return type None

key_partition(*topic: str*, *key: bytes*) → faust.types.tuples.TP
Hash key to determine partition.

Return type *TP*

async begin_transaction(*transactional_id: str*) → None
Begin transaction by id.

Return type None

async commit_transaction(*transactional_id: str*) → None
Commit transaction by id.

Return type None

async abort_transaction(*transactional_id: str*) → None
Abort and rollback transaction by id.

Return type None

async stop_transaction(*transactional_id: str*) → None
Stop transaction by id.

Return type None

async maybe_begin_transaction(*transactional_id: str*) → None
Begin transaction by id, if not already started.

Return type None

async commit_transactions(*tid_to_offset_map: Mapping[str, Mapping[faust.types.tuples.TP, int]]*, *group_id: str*, *start_new_transaction: bool = True*) → None
Commit transactions.

Return type None

logger = <Logger faust.transport.producer (WARNING)>

supports_headers() → bool
Return True if headers are supported by this transport.

Return type bool

class faust.transport.base.**Transport**(*url: List[yarl.URL]*, *app: faust.types.app.AppT*, *loop: asyncio.events.AbstractEventLoop = None*) → None
Message transport implementation.

class Consumer(*transport: faust.types.transports.TransportT, callback: Callable[faust.types.tuples.Message, Awaitable], on_partitions_revoked: Callable[Set[faust.types.tuples.TP], Awaitable[None]], on_partitions_assigned: Callable[Set[faust.types.tuples.TP], Awaitable[None]], *, commit_interval: float = None, commit_livelock_soft_timeout: float = None, loop: asyncio.events.AbstractEventLoop = None, **kwargs: Any*) → None

Base Consumer.

ack(*message: faust.types.tuples.Message*) → bool

Mark message as being acknowledged by stream.

Return type bool

close() → None

Close consumer for graceful shutdown.

Return type None

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]] = None*, *start_new_transaction: bool = True*) → bool

Maybe commit the offset for all or specific topics.

Parameters **topics** (Optional[AbstractSet[Union[str, *TP*]]]) – Set containing topics and/or TopicPartitions to commit.

Return type bool

async commit_and_end_transactions() → None

Commit all safe offsets and end transaction.

Return type None

consumer_stopped_errors = ()

flow_active = True

async force_commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]] = None*, *start_new_transaction: bool = True*) → bool

Force offset commit.

Return type bool

getmany(*timeout: float*) → AsyncIterator[Tuple[faust.types.tuples.TP, faust.types.tuples.Message]]

Fetch batch of messages from server.

Return type AsyncIterator[Tuple[*TP*, *Message*]]

logger = <Logger faust.transport.consumer (WARNING)>

async maybe_wait_for_commit_to_finish() → bool

Wait for any existing commit operation to finish.

Return type bool

on_init_dependencies() → Iterable[mode.types.services.ServiceT]

Return list of services this consumer depends on.

Return type Iterable[ServiceT[]]

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None

Call during rebalancing when partitions are being assigned.

Return type None

async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None

Call during rebalancing when partitions are being revoked.

Return type None

async on_restart() → None

Call when the consumer is restarted.

Return type None

async on_stop() → None
Call when consumer is stopping.
Return type None

async on_task_error(*exc: BaseException*) → None
Call when processing a message failed.
Return type None

pause_partitions(*tps: Iterable[faust.types.tuples.TP]*) → None
Pause fetching from partitions.
Return type None

async perform_seek() → None
Seek all partitions to their current committed position.
Return type None

resume_flow() → None
Allow consumer to process messages.
Return type None

resume_partitions(*tps: Iterable[faust.types.tuples.TP]*) → None
Resume fetching from partitions.
Return type None

async seek(*partition: faust.types.tuples.TP*, *offset: int*) → None
Seek partition to specific offset.
Return type None

abstract async seek_to_committed() → Mapping[faust.types.tuples.TP, int]
Seek all partitions to their committed offsets.
Return type Mapping[*TP*, int]

stop_flow() → None
Block consumer from processing any more messages.
Return type None

track_message(*message: faust.types.tuples.Message*) → None
Track message and mark it as pending ack.
Return type None

property unacked
Return the set of currently unacknowledged messages. :rtype: Set[*Message*]

async wait_empty() → None
Wait for all messages that started processing to be acked.
Return type None

class Producer(*transport: faust.types.transports.TransportT*, *loop: asyncio.events.AbstractEventLoop = None*, ***kwargs: Any*) → None
Base Producer.

async abort_transaction(*transactional_id: str*) → None
Abort and rollback transaction by id.
Return type None

async begin_transaction(*transactional_id: str*) → None
Begin transaction by id.
Return type None

async commit_transaction(*transactional_id: str*) → None
Commit transaction by id.
Return type None

async commit_transactions(*tid_to_offset_map: Mapping[str, Mapping[faust.types.tuples.TP, int]], group_id: str, start_new_transaction: bool = True*) → None

Commit transactions.

Return type None

async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 1000.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None

Create/declare topic on server.

Return type None

async flush() → None

Flush all in-flight messages.

Return type None

key_partition(*topic: str*, *key: bytes*) → faust.types.tuples.TP

Hash key to determine partition.

Return type *TP*

logger = <Logger faust.transport.producer (WARNING)>

async maybe_begin_transaction(*transactional_id: str*) → None

Begin transaction by id, if not already started.

Return type None

async on_start() → None

Service is starting.

Return type None

async send(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]

Schedule message to be sent by producer.

Return type Awaitable[*RecordMetadata*]

async send_and_wait(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → faust.types.tuples.RecordMetadata

Send message and wait for it to be transmitted.

Return type *RecordMetadata*

send_soon(*fut: faust.types.tuples.FutureMessage*) → None

Return type None

async stop_transaction(*transactional_id: str*) → None

Stop transaction by id.

Return type None

supports_headers() → bool

Return `True` if headers are supported by this transport.

Return type bool

class TransactionManager(*transport: faust.types.transports.TransportT*, ***, *consumer: faust.types.transports.ConsumerT*, *producer: faust.types.transports.ProducerT*, ***kwargs: Any*) → None

Manage producer transactions.

async commit(*offsets: Mapping[faust.types.tuples.TP, int], start_new_transaction: bool = True*) → bool
Commit offsets for partitions.
Return type bool

async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 30.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None
Create/declare topic on server.
Return type None

async flush() → None
Wait for producer to transmit all pending messages.
Return type None

key_partition(*topic: str*, *key: bytes*) → faust.types.tuples.TP
Return type *TP*

logger = <Logger faust.transport.consumer (WARNING)>

async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call when the cluster is rebalancing and partitions are revoked.
Return type None

async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None
Call when the cluster is rebalancing.
Return type None

async send(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]
Schedule message to be sent by producer.
Return type Awaitable[*RecordMetadata*]

async send_and_wait(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → faust.types.tuples.RecordMetadata
Send message and wait for it to be transmitted.
Return type *RecordMetadata*

supports_headers() → bool
Return `True` if the Kafka server supports headers.
Return type bool

transactional_id_format = '{group_id}-{tpg.group}-{tpg.partition}'

class Conductor(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Manages the channels that subscribe to topics.

- Consumes messages from topic using a single consumer.
- Forwards messages to all channels subscribing to a topic.

acks_enabled_for(*topic: str*) → bool
Return `True` if acks are enabled for topic by name.
Return type bool

add(*topic: Any*) → None
Register topic to be subscribed.
Return type None

clear() → None
Clear all subscriptions.
Return type None

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]]*) → bool
Commit offsets in topics.
Return type bool

discard(*topic: Any*) → None
Unregister topic from conductor.
Return type None

property label
Return label for use in logs. :rtype: str

logger = <Logger faust.transport.conductor (WARNING)>

async maybe_wait_for_subscriptions() → None
Return type None

async on_client_only_start() → None
Return type None

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing and partitions are assigned.
Return type None

property shortlabel
Return short label for use in logs. :rtype: str

async wait_for_subscriptions() → None
Wait for consumer to be subscribed.
Return type None

class Fetcher(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Service fetching messages from Kafka.

logger = <Logger faust.transport.consumer (WARNING)>

async on_stop() → None
Call when the fetcher is stopping.
Return type None

create_consumer(*callback: Callable[faust.types.tuples.Message, Awaitable], **kwargs: Any*) → faust.types.transports.ConsumerT
Create new consumer.

Return type *ConsumerT*[]

create_producer(***kwargs: Any*) → faust.types.transports.ProducerT
Create new producer.

Return type *ProducerT*[]

create_transaction_manager(*consumer: faust.types.transports.ConsumerT*, *producer: faust.types.transports.ProducerT*, ***kwargs: Any*) → faust.types.transports.TransactionManagerT
Create new transaction manager.

Return type *TransactionManagerT*[]

create_conductor(***kwargs: Any*) → faust.types.transports.ConductorT
Create new consumer conductor.

Return type *ConductorT*[]

faust.transport.conductor

The conductor delegates messages from the consumer to the streams.

class faust.transport.conductor.**ConductorCompiler**
Compile a function to handle the messages for a topic+partition.

build(*conductor: faust.transport.conductor.Conductor, tp: faust.types.tuples.TP, channels: MutableSet[faust.transport.conductor._Topic]*) → Callable[faust.types.tuples.Message, Awaitable]
Generate closure used to deliver messages.

Return type Callable[[*Message*], Awaitable[+T_co]]

class faust.transport.conductor.**Conductor**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Manages the channels that subscribe to topics.

- Consumes messages from topic using a single consumer.
- Forwards messages to all channels subscribing to a topic.

logger = <Logger faust.transport.conductor (WARNING)>

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]]*) → bool
Commit offsets in topics.

Return type bool

acks_enabled_for(*topic: str*) → bool
Return True if acks are enabled for topic by name.

Return type bool

async wait_for_subscriptions() → None
Wait for consumer to be subscribed.

Return type None

async maybe_wait_for_subscriptions() → None

Return type None

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call when cluster is rebalancing and partitions are assigned.

Return type None

async on_client_only_start() → None

Return type None

clear() → None
Clear all subscriptions.

Return type None

add(*topic: Any*) → None
Register topic to be subscribed.

Return type None

discard(*topic: Any*) → None
Unregister topic from conductor.

Return type None

property label
Return label for use in logs. :rtype: str

property shortlabel
Return short label for use in logs. :rtype: str

faust.transport.consumer

Consumer - fetching messages and managing consumer state.

The Consumer is responsible for:

- Holds reference to the transport that created it
- … and the app via `self.transport.app`.
- Has a callback that usually points back to `Conductor.on_message`.
- Receives messages and calls the callback for every message received.
- Keeps track of the message and its acked/unacked status.
- The Conductor forwards the message to all Streams that subscribes to the topic the message was sent to.
  - Messages are reference counted, and the Conductor increases the reference count to the number of subscribed streams.
  - `Stream.__aiter__` is set up in a way such that when what is iterating over the stream is finished with the message, a finally: block will decrease the reference count by one.
  - When the reference count for a message hits zero, the stream will call `Consumer.ack(message)`, which will mark that topic + partition + offset combination as "committable"
  - If all the streams share the same key_type/value_type, the conductor will only deserialize the payload once.
- Commits the offset at an interval
  - The Consumer has a background thread that periodically commits the offset.
  - If the consumer marked an offset as committable this thread will advance the committed offset.
  - To find the offset that it can safely advance to the commit thread will traverse the _acked mapping of TP to list of acked offsets, by finding a range of consecutive acked offsets (see note in _new_offset).

class faust.transport.consumer.**Fetcher**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Service fetching messages from Kafka.

logger = <Logger faust.transport.consumer (WARNING)>

async on_stop() → None
Call when the fetcher is stopping.

Return type None

class faust.transport.consumer.**Consumer**(*transport: faust.types.transports.TransportT, callback: Callable[faust.types.tuples.Message, Awaitable], on_partitions_revoked: Callable[Set[faust.types.tuples.TP], Awaitable[None]], on_partitions_assigned: Callable[Set[faust.types.tuples.TP], Awaitable[None]], *, commit_interval: float = None, commit_livelock_soft_timeout: float = None, loop: asyncio.events.AbstractEventLoop = None, **kwargs: Any*) → None

Base Consumer.

logger = <Logger faust.transport.consumer (WARNING)>

consumer_stopped_errors = ()

Tuple of exception types that may be raised when the underlying consumer driver is stopped.

flow_active = True

on_init_dependencies() → Iterable[mode.types.services.ServiceT]

Return list of services this consumer depends on.

Return type Iterable[ServiceT[]]

async on_restart() → None

Call when the consumer is restarted.

Return type None

async perform_seek() → None

Seek all partitions to their current committed position.

Return type None

abstract async seek_to_committed() → Mapping[faust.types.tuples.TP, int]

Seek all partitions to their committed offsets.

Return type Mapping[*TP*, int]

async seek(*partition: faust.types.tuples.TP*, *offset: int*) → None

Seek partition to specific offset.

Return type None

stop_flow() → None

Block consumer from processing any more messages.

Return type None

resume_flow() → None

Allow consumer to process messages.

Return type None

pause_partitions(*tps: Iterable[faust.types.tuples.TP]*) → None

Pause fetching from partitions.

Return type None

resume_partitions(*tps: Iterable[faust.types.tuples.TP]*) → None

Resume fetching from partitions.

Return type None

async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None
Call during rebalancing when partitions are being revoked.

Return type None

async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None
Call during rebalancing when partitions are being assigned.

Return type None

getmany(*timeout: float*) → AsyncIterator[Tuple[faust.types.tuples.TP, faust.types.tuples.Message]]
Fetch batch of messages from server.

Return type AsyncIterator[Tuple[*TP*, *Message*]]

track_message(*message: faust.types.tuples.Message*) → None
Track message and mark it as pending ack.

Return type None

ack(*message: faust.types.tuples.Message*) → bool
Mark message as being acknowledged by stream.

Return type bool

async wait_empty() → None
Wait for all messages that started processing to be acked.

Return type None

async commit_and_end_transactions() → None
Commit all safe offsets and end transaction.

Return type None

async on_stop() → None
Call when consumer is stopping.

Return type None

async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]] = None*, *start_new_transaction: bool = True*) → bool
Maybe commit the offset for all or specific topics.

Parameters **topics** (Optional[AbstractSet[Union[str, *TP*]]]) – Set containing topics and/or TopicPartitions to commit.

Return type bool

async maybe_wait_for_commit_to_finish() → bool
Wait for any existing commit operation to finish.

Return type bool

async force_commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]] = None*, *start_new_transaction: bool = True*) → bool
Force offset commit.

Return type bool

async on_task_error(*exc: BaseException*) → None
Call when processing a message failed.

Return type None

close() → None
Close consumer for graceful shutdown.

Return type None

property unacked
Return the set of currently unacknowledged messages. :rtype: Set[*Message*]

faust.transport.producer

Producer.

The Producer is responsible for:

- Holds reference to the transport that created it
- … and the app via `self.transport.app`.
- Sending messages.

class faust.transport.producer.**Producer**(*transport: faust.types.transports.TransportT*, *loop: asyncio.events.AbstractEventLoop = None*, ***kwargs: Any*) → None
Base Producer.

async on_start() → None
Service is starting.

Return type None

async send(*topic: str*, *key: Optional[bytes]*, *value: Optional[bytes]*, *partition: Optional[int]*, *timestamp: Optional[float]*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]*, ***, *transactional_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]
Schedule message to be sent by producer.

Return type Awaitable[*RecordMetadata*]

send_soon(*fut: faust.types.tuples.FutureMessage*) → None

Return type None

async send_and_wait(*topic: str*, *key: Optional[bytes]*, *value: Optional[bytes]*, *partition: Optional[int]*, *timestamp: Optional[float]*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None]*, ***, *transactional_id: str = None*) → faust.types.tuples.RecordMetadata
Send message and wait for it to be transmitted.

Return type *RecordMetadata*

async flush() → None
Flush all in-flight messages.

Return type None

async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 1000.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None
Create/declare topic on server.

Return type None

key_partition(*topic: str*, *key: bytes*) → faust.types.tuples.TP
Hash key to determine partition.

Return type *TP*

async begin_transaction(*transactional_id: str*) → None
Begin transaction by id.

Return type None

async commit_transaction(*transactional_id: str*) → None
Commit transaction by id.

Return type None

async abort_transaction(*transactional_id: str*) → None
Abort and rollback transaction by id.

Return type None

async stop_transaction(*transactional_id: str*) → None
Stop transaction by id.

Return type None

async maybe_begin_transaction(*transactional_id: str*) → None
Begin transaction by id, if not already started.

Return type None

async commit_transactions(*tid_to_offset_map: Mapping[str, Mapping[faust.types.tuples.TP, int]], group_id: str, start_new_transaction: bool = True*) → None
Commit transactions.

Return type None

logger = <Logger faust.transport.producer (WARNING)>

supports_headers() → bool
Return True if headers are supported by this transport.

Return type bool

faust.transport.drivers

Transport registry.

faust.transport.drivers.**by_name**(*name: Union[_T, str]*) → _T

Return type ~_T

faust.transport.drivers.**by_url**(*url: Union[str, yarl.URL]*) → _T
Get class associated with URL (scheme is used as alias key).

Return type ~_T

faust.transport.drivers.aiokafka

Message transport using aiokafka.

class faust.transport.drivers.aiokafka.**Consumer**(**args: Any, **kwargs: Any*) → None
Kafka consumer using aiokafka.

logger = <Logger faust.transport.drivers.aiokafka (WARNING)>

RebalanceListener
alias of ConsumerRebalanceListener

consumer_stopped_errors = (<class 'aiokafka.errors.ConsumerStoppedError'>,)

async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 30.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None

Create/declare topic on server.

Return type None

async on_stop() → None

Call when consumer is stopping.

Return type None

class faust.transport.drivers.aiokafka.**Producer**(*transport: faust.types.transports.TransportT*, *loop: asyncio.events.AbstractEventLoop = None*, ***kwargs: Any*) → None

Kafka producer using aiokafka.

logger = <Logger faust.transport.drivers.aiokafka (WARNING)>

allow_headers = True

async begin_transaction(*transactional_id: str*) → None

Begin transaction by id.

Return type None

async commit_transaction(*transactional_id: str*) → None

Commit transaction by id.

Return type None

async abort_transaction(*transactional_id: str*) → None

Abort and rollback transaction by id.

Return type None

async stop_transaction(*transactional_id: str*) → None

Stop transaction by id.

Return type None

async maybe_begin_transaction(*transactional_id: str*) → None

Begin transaction (if one does not already exist).

Return type None

async commit_transactions(*tid_to_offset_map: Mapping[str, Mapping[faust.types.tuples.TP, int]]*, *group_id: str*, *start_new_transaction: bool = True*) → None

Commit transactions.

Return type None

async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 20.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None

Create/declare topic on server.

Return type None

async on_start() → None

Call when producer starts.

Return type None

async on_stop() → None
Call when producer stops.

Return type None

async send(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]
Schedule message to be transmitted by producer.

Return type Awaitable[*RecordMetadata*]

async send_and_wait(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → faust.types.tuples.RecordMetadata
Send message and wait for it to be transmitted.

Return type *RecordMetadata*

async flush() → None
Wait for producer to finish transmitting all buffered messages.

Return type None

key_partition(*topic: str*, *key: bytes*) → faust.types.tuples.TP
Hash key to determine partition destination.

Return type *TP*

supports_headers() → bool
Return True if message headers are supported.

Return type bool

class faust.transport.drivers.aiokafka.**Transport**(**args: Any*, ***kwargs: Any*) → None
Kafka transport using aiokafka.

class Consumer(**args: Any*, ***kwargs: Any*) → None
Kafka consumer using aiokafka.

RebalanceListener
alias of ConsumerRebalanceListener

consumer_stopped_errors = (<class 'aiokafka.errors.ConsumerStoppedError'>,)

async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 30.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None
Create/declare topic on server.

Return type None

logger = <Logger faust.transport.drivers.aiokafka (WARNING)>

async on_stop() → None
Call when consumer is stopping.

Return type None

class Producer(*transport: faust.types.transports.TransportT*, *loop: asyncio.events.AbstractEventLoop = None*, ***kwargs: Any*) → None
Kafka producer using aiokafka.

async abort_transaction(*transactional_id: str*) → None
Abort and rollback transaction by id.

Return type None

allow_headers = True

async begin_transaction(*transactional_id: str*) → None
Begin transaction by id.
Return type None

async commit_transaction(*transactional_id: str*) → None
Commit transaction by id.
Return type None

async commit_transactions(*tid_to_offset_map: Mapping[str, Mapping[faust.types.tuples.TP, int]], group_id: str, start_new_transaction: bool = True*) → None
Commit transactions.
Return type None

async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 20.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None
Create/declare topic on server.
Return type None

async flush() → None
Wait for producer to finish transmitting all buffered messages.
Return type None

key_partition(*topic: str*, *key: bytes*) → faust.types.tuples.TP
Hash key to determine partition destination.
Return type *TP*

logger = <Logger faust.transport.drivers.aiokafka (WARNING)>

async maybe_begin_transaction(*transactional_id: str*) → None
Begin transaction (if one does not already exist).
Return type None

async on_start() → None
Call when producer starts.
Return type None

async on_stop() → None
Call when producer stops.
Return type None

async send(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]
Schedule message to be transmitted by producer.
Return type Awaitable[*RecordMetadata*]

async send_and_wait(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → faust.types.tuples.RecordMetadata
Send message and wait for it to be transmitted.
Return type *RecordMetadata*

async stop_transaction(*transactional_id: str*) → None
Stop transaction by id.
Return type None

supports_headers() → bool
Return True if message headers are supported.
Return type bool

default_port = 9092

driver_version = 'aiokafka=1.1.3'

faust.transport.utils

Transport utils - scheduling.

faust.transport.utils.**TopicIndexMap**
alias of typing.MutableMapping

class faust.transport.utils.**DefaultSchedulingStrategy**
Consumer record scheduler.

Delivers records in round robin between both topics and partitions.

classmethod map_from_records(*records: Mapping[faust.types.tuples.TP, List]*) → MutableMapping[str, faust.transport.utils.TopicBuffer]
Convert records to topic index map.

Return type MutableMapping[str, *TopicBuffer*[]]

iterate(*records: Mapping[faust.types.tuples.TP, List]*) → Iterator[Tuple[faust.types.tuples.TP, Any]]
Iterate over records in round-robin order.

Return type Iterator[Tuple[*TP*, Any]]

records_iterator(*index: MutableMapping[str, TopicBuffer]*) → Iterator[Tuple[faust.types.tuples.TP, Any]]
Iterate over topic index map in round-robin order.

Return type Iterator[Tuple[*TP*, Any]]

class faust.transport.utils.**TopicBuffer** → None
Data structure managing the buffer for incoming records in a topic.

add(*tp: faust.types.tuples.TP*, *buffer: List*) → None
Add topic partition buffer to the cycle.

Return type None

1.6.12 Assignor

faust.assignor.client_assignment

Client Assignment.

class faust.assignor.client_assignment.**CopartitionedAssignment**(*actives: Set[int] = None*, *standbys: Set[int] = None*, *topics: Set[str] = None*) → None

Copartitioned Assignment.

validate() → None

Return type None

num_assigned(*active: bool*) → int

Return type int

get_unassigned(*num_partitions: int*, *active: bool*) → Set[int]

Return type Set[int]

pop_partition(*active: bool*) → int

Return type int

unassign_partition(*partition: int*, *active: bool*) → None

Return type None

assign_partition(*partition: int*, *active: bool*) → None

Return type None

unassign_extras(*capacity: int*, *replicas: int*) → None

Return type None

partition_assigned(*partition: int*, *active: bool*) → bool

Return type bool

promote_standby_to_active(*standby_partition: int*) → None

Return type None

get_assigned_partitions(*active: bool*) → Set[int]

Return type Set[int]

can_assign(*partition: int*, *active: bool*) → bool

Return type bool

class faust.assignor.client_assignment.**ClientAssignment**(*actives*, *standbys*, ***, *__strict__=True*, *__faust=None*, ***kwargs*) → None

Client Assignment data model.

actives

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

standbys

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

property active_tps

Return type Set[*TP*]

property standby_tps

Return type Set[*TP*]

kafka_protocol_assignment(*table_manager: faust.types.tables.TableManagerT*) → Sequence[Tuple[str, List[int]]]

Return type Sequence[Tuple[str, List[int]]]

add_copartitioned_assignment(*assignment: faust.assignor.client_assignment.CopartitionedAssignment*) → None

Return type None

copartitioned_assignment(*topics: Set[str]*) → faust.assignor.client_assignment.CopartitionedAssignment

Return type *CopartitionedAssignment*

asdict()

class faust.assignor.client_assignment.**ClientMetadata**(*assignment*, *url*, *changelog_distribution*, *topic_groups=None*, ***, *__strict__=True*, *__faust=None*, ***kwargs*) → None

Client Metadata data model.

assignment

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

url

asdict()

changelog_distribution
Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

topic_groups

faust.assignor.cluster_assignment

Cluster assignment.

faust.assignor.cluster_assignment.**CopartMapping**
alias of typing.MutableMapping

class faust.assignor.cluster_assignment.**ClusterAssignment**(*subscriptions=None*, *assignments=None*, ***, *__strict__=True*, *__faust=None*, ***kwargs*) → None

Cluster assignment state.

subscriptions
Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

assignments

Describes a field.

Used for every field in Record so that they can be used in join's /group_by etc.

Examples

```
>>> class Withdrawal(Record):
...     account_id: str
...     amount: float = 0.0
```

```
>>> Withdrawal.account_id
<FieldDescriptor: Withdrawal.account_id: str>
>>> Withdrawal.amount
<FieldDescriptor: Withdrawal.amount: float = 0.0>
```

Parameters

- **field** (*str*) – Name of field.
- **type** (*Type*) – Field value type.
- **required** (*bool*) – Set to false if field is optional.
- **default** (*Any*) – Default value when *required=False*.

Keyword Arguments

- **model** (*Type*) – Model class the field belongs to.
- **parent** (FieldDescriptorT) – parent field if any.

topics() → Set[str]

Return type Set[str]

add_client(*client: str, subscription: List[str], metadata: faust.assignor.client_assignment.ClientMetadata*) → None

Return type None

copartitioned_assignments(*copartitioned_topics: Set[str]*) → MutableMapping[str, faust.assignor.client_assignment.CopartitionedAssignment]

Return type MutableMapping[str, *CopartitionedAssignment*]

asdict()

faust.assignor.copartitioned_assignor

Copartitioned Assignor.

class faust.assignor.copartitioned_assignor.**CopartitionedAssignor**(*topics: Iterable[str], cluster_asgn: MutableMapping[str, faust.assignor.client_assignment.Copa num_partitions: int, replicas: int, capacity: int = None*) → None

Copartitioned Assignor.

All copartitioned topics must have the same number of partitions

The assignment is sticky which uses the following heuristics:

- Maintain existing assignments as long as within capacity for each client
- Assign actives to standbys when possible (within capacity)
- Assign in order to fill capacity of the clients

We optimize for not over utilizing resources instead of under-utilizing resources. This results in a balanced assignment when capacity is the default value which is `ceil(num partitions / num clients)`

Notes

Currently we raise an exception if number of clients is not enough for the desired *replication*.

get_assignment() → MutableMapping[str, faust.assignor.client_assignment.CopartitionedAssignment]

Return type MutableMapping[str, *CopartitionedAssignment*]

faust.assignor.leader_assignor

Leader assignor.

class faust.assignor.leader_assignor.**LeaderAssignor**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None

Leader assignor, ensures election of a leader.

async on_start() → None
Service is starting.

Return type None

is_leader() → bool

Return type bool

logger = <Logger faust.assignor.leader_assignor (WARNING)>

faust.assignor.partition_assignor

Partition assignor.

faust.assignor.partition_assignor.**MemberAssignmentMapping**
alias of typing.MutableMapping

faust.assignor.partition_assignor.**MemberMetadataMapping**
alias of typing.MutableMapping

faust.assignor.partition_assignor.**MemberSubscriptionMapping**
alias of typing.MutableMapping

faust.assignor.partition_assignor.**ClientMetadataMapping**
alias of typing.MutableMapping

faust.assignor.partition_assignor.**ClientAssignmentMapping**
alias of typing.MutableMapping

faust.assignor.partition_assignor.**CopartitionedGroups**
alias of typing.MutableMapping

class faust.assignor.partition_assignor.**PartitionAssignor**(*app: faust.types.app.AppT*, *replicas: int = 0*) → None

PartitionAssignor handles internal topic creation.

Further, this assignor needs to be sticky and potentially redundant

Notes

Interface copied from kafka.coordinator.assignors.abstract.

group_for_topic(*topic: str*) → int

Return type int

property changelog_distribution

Return type MutableMapping[str, MutableMapping[str, List[int]]]

on_assignment(*assignment: kafka.coordinator.protocol.ConsumerProtocolMemberMetadata*) → None
Callback that runs on each assignment.

This method can be used to update internal state, if any, of the partition assignor.

Parameters **assignment** (*MemberAssignment*) – the member's assignment

Return type None

metadata(*topics: Set[str]*) → kafka.coordinator.protocol.ConsumerProtocolMemberMetadata
Generate ProtocolMetadata to be submitted via JoinGroupRequest.

Parameters **topics** (*set*) – a member's subscribed topics

Return type ConsumerProtocolMemberMetadata

Returns MemberMetadata struct

assign(*cluster: kafka.cluster.ClusterMetadata, member_metadata: MutableMapping[str, kafka.coordinator.protocol.ConsumerProtocolMemberMetadata]*) → MutableMapping[str, kafka.coordinator.protocol.ConsumerProtocolMemberAssignment]
Perform group assignment given cluster metadata and member subscriptions

Parameters

- **cluster** (*ClusterMetadata*) – metadata for use in assignment
- **(dict of {member_id** (*members*) – MemberMetadata}): decoded metadata for each member in the group.

Return type MutableMapping[str, ConsumerProtocolMemberAssignment]

Returns {member_id: MemberAssignment}

Return type dict

property name
.name should be a string identifying the assignor :rtype: str

property version

Return type int

assigned_standbys() → Set[faust.types.tuples.TP]

Return type Set[*TP*]

assigned_actives() → Set[faust.types.tuples.TP]

Return type Set[*TP*]

table_metadata(*topic: str*) → MutableMapping[str, MutableMapping[str, List[int]]]

Return type MutableMapping[str, MutableMapping[str, List[int]]]

tables_metadata() → MutableMapping[str, MutableMapping[str, List[int]]]

Return type MutableMapping[str, MutableMapping[str, List[int]]]

key_store(*topic: str*, *key: bytes*) → yarl.URL

Return type URL

is_active(*tp: faust.types.tuples.TP*) → bool

Return type bool

is_standby(*tp: faust.types.tuples.TP*) → bool

Return type bool

1.6.13 Types

faust.types.agents

faust.types.agents.**AgentErrorHandler**
alias of typing.Callable

faust.types.agents.**AgentFun**
alias of typing.Callable

faust.types.agents.**SinkT = typing.Union[_ForwardRef('AgentT'), faust.types.channels.Channel**
Agent, Channel or callable/async callable taking value as argument.

Type A sink can be

class faust.types.agents.**ActorT**(*agent: faust.types.agents.AgentT*, *stream: faust.types.streams.StreamT*, *it: _T*, *active_partitions: Set[faust.types.tuples.TP] = None*, ***kwargs: Any*) → None

index = None
If multiple instance are started for concurrency, this is its index.

abstract cancel() → None

Return type None

abstract async on_isolated_partition_revoked(*tp: faust.types.tuples.TP*) → None

Return type None

abstract async on_isolated_partition_assigned(*tp: faust.types.tuples.TP*) → None

Return type None

class faust.types.agents.**AsyncIterableActorT**(*agent: faust.types.agents.AgentT*, *stream: faust.types.streams.StreamT*, *it: _T*, *active_partitions: Set[faust.types.tuples.TP] = None*, ***kwargs: Any*) → None
Used for agent function that yields.

class faust.types.agents.**AwaitableActorT**(*agent: faust.types.agents.AgentT*, *stream: faust.types.streams.StreamT*, *it: _T*, *active_partitions: Set[faust.types.tuples.TP] = None*, ***kwargs: Any*) → None
Used for agent function that do not yield.

faust.types.agents.**ActorRefT**
alias of *faust.types.agents.ActorT*

class faust.types.agents.**AgentT**(*fun: Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], *, name: str = None, app: faust.types.agents._AppT = None, channel: Union[str, faust.types.channels.ChannelT] = None, concurrency: int = 1, sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None, on_error: Callable[[AgentT, BaseException], Awaitable] = None, supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None, help: str = None, schema: faust.types.serializers.SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, isolated_partitions: bool = False, **kwargs: Any*) → None

abstract test_context(*channel: faust.types.channels.ChannelT = None*, *supervisor_strategy: mode.types.supervisors.SupervisorStrategyT = None*, ***kwargs: Any*) → faust.types.agents.AgentTestWrapperT

Return type *AgentTestWrapperT*[]

abstract add_sink(*sink: Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]*) → None

Return type None

abstract stream(***kwargs: Any*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async cast(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*) → None

Return type None

abstract async ask(*value: Union[bytes, faust.types.core._ModelT, Any] = None, *, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None*) → Any

Return type Any

abstract async send(**, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract async map(*values: Union[AsyncIterable, Iterable], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator

abstract async kvmap(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → AsyncIterator[str]

abstract async join(*values: Union[AsyncIterable[Union[bytes, faust.types.core._ModelT, Any]], Iterable[Union[bytes, faust.types.core._ModelT, Any]]], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]

Return type List[Any]

abstract async kvjoin(*items: Union[AsyncIterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]], Iterable[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], Union[bytes, faust.types.core._ModelT, Any]]]], reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None*) → List[Any]

Return type List[Any]

abstract info() → Mapping

Return type Mapping[~KT, +VT_co]

abstract clone(*, *cls: Type[AgentT] = None*, ***kwargs: Any*) → faust.types.agents.AgentT

Return type *AgentT*[]

abstract get_topic_names() → Iterable[str]

Return type Iterable[str]

abstract property channel

Return type *ChannelT*[]

abstract property channel_iterator

Return type AsyncIterator[+T_co]

class faust.types.agents.**AgentManagerT**(*, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None

abstract async on_rebalance(*revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None

Return type None

class faust.types.agents.**AgentTestWrapperT**(**args: Any, original_channel: faust.types.channels.ChannelT = None, **kwargs: Any*) → None

sent_offset = 0

processed_offset = 0

abstract async put(*value: Union[bytes, faust.types.core._ModelT, Any] = None, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, *, reply_to: Union[AgentT, faust.types.channels.ChannelT, str] = None, correlation_id: str = None, wait: bool = True*) → faust.types.events.EventT

Return type *EventT*[]

abstract to_message(*key: Union[bytes, faust.types.core._ModelT, Any, None], value: Union[bytes, faust.types.core._ModelT, Any], *, partition: int = 0, offset: int = 0, timestamp: float = None, timestamp_type: int = 0, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*) → faust.types.tuples.Message

Return type *Message*

abstract async throw(*exc: BaseException*) → None

Return type None

faust.types.app

class faust.types.app.**AppT**(*id: str*, *, *monitor: faust.types.app._Monitor*, *config_source: Any = None*, ***options: Any*) → None

Abstract type for the Faust application.

See also:

faust.App.

finalized = False

Set to true when the app is finalized (can read configuration).

configured = False

Set to true when the app has read configuration.

rebalancing = False

Set to true if the worker is currently rebalancing.

rebalancing_count = 0

unassigned = False

in_worker = False

on_configured(**args: Any*, ***kwargs: Any*) → None **= <SyncSignal: AppT.on_configured>**

on_before_configured(**args: Any*, ***kwargs: Any*) → None **= <SyncSignal: AppT.on_before_configured>**

on_after_configured(**args: Any*, ***kwargs: Any*) → None **= <SyncSignal: AppT.on_after_configured>**

on_partitions_assigned(**args: Any*, ***kwargs: Any*) → None **= <Signal: AppT.on_partitions_assigned>**

on_partitions_revoked(**args: Any*, ***kwargs: Any*) → None **= <Signal: AppT.on_partitions_revoked>**

on_rebalance_complete(**args: Any*, ***kwargs: Any*) → None **= <Signal: AppT.on_rebalance_complete>**

on_before_shutdown(**args: Any*, ***kwargs: Any*) → None **= <Signal: AppT.on_before_shutdown>**

on_worker_init(**args: Any*, ***kwargs: Any*) → None **= <SyncSignal: AppT.on_worker_init>**

on_produce_message(**args: Any*, ***kwargs: Any*) → None **= <SyncSignal: AppT.on_produce_message>**

abstract config_from_object(*obj: Any*, ***, *silent: bool = False*, *force: bool = False*) → None

Return type None

abstract finalize() → None

Return type None

abstract main() → NoReturn

Return type _NoReturn

abstract worker_init() → None

Return type None

abstract discover(**extra_modules: str*, *categories: Iterable[str] = ('a', 'b', 'c')*, *ignore: Iterable[Any] = ('foo', 'bar')*) → None

Return type None

abstract topic(**topics: str*, *pattern: Union[str, Pattern[~AnyStr]] = None*, *schema: faust.types.app._SchemaT = None*, *key_type: faust.types.app._ModelArg = None*, *value_type: faust.types.app._ModelArg = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *partitions: int = None*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *replicas: int = None*, *acks: bool = True*, *internal: bool = False*, *config: Mapping[str, Any] = None*, *maxsize: int = None*, *allow_empty: bool = False*, *has_prefix: bool = False*, *loop: asyncio.events.AbstractEventLoop = None*) → faust.types.topics.TopicT

Return type *TopicT*[]

abstract channel(***, *schema: faust.types.app._SchemaT = None*, *key_type: faust.types.app._ModelArg = None*, *value_type: faust.types.app._ModelArg = None*, *maxsize: int = None*, *loop: asyncio.events.AbstractEventLoop = None*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract agent(*channel: Union[str, faust.types.channels.ChannelT] = None*, ***, *name: str = None*, *concurrency: int = 1*, *supervisor_strategy: Type[mode.types.supervisors.SupervisorStrategyT] = None*, *sink: Iterable[Union[AgentT, faust.types.channels.ChannelT, Callable[Any, Optional[Awaitable]]]] = None*, *isolated_partitions: bool = False*, *use_reply_headers: bool = False*, ***kwargs: Any*) → Callable[Callable[faust.types.streams.StreamT, Union[Coroutine[[Any, Any], None], Awaitable[None], AsyncIterable]], faust.types.agents.AgentT]

Return type Callable[[Callable[[*StreamT*[+T_co]], Union[Coroutine[Any, Any, None], Awaitable[None], AsyncIterable[+T_co]]]], *AgentT*[]]

abstract task(*fun: Union[Callable[AppT, Awaitable], Callable[Awaitable]]*, ***, *on_leader: bool = False*, *traced: bool = True*) → Callable

abstract timer(*interval: Union[datetime.timedelta, float, str], on_leader: bool = False, traced: bool = True, name: str = None, max_drift_correction: float = 0.1*) → Callable

Return type Callable

abstract crontab(*cron_format: str, *, timezone: datetime.tzinfo = None, on_leader: bool = False, traced: bool = True*) → Callable

Return type Callable

abstract service(*cls: Type[mode.types.services.ServiceT]*) → Type[mode.types.services.ServiceT]

Return type Type[ServiceT[]]

abstract stream(*channel: AsyncIterable, beacon: mode.utils.types.trees.NodeT = None, **kwargs: Any*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract Table(*name: str, *, default: Callable[Any] = None, window: faust.types.windows.WindowT = None, partitions: int = None, help: str = None, **kwargs: Any*) → faust.types.tables.TableT

Return type *TableT*[~KT, ~VT]

abstract GlobalTable(*name: str, *, default: Callable[Any] = None, window: faust.types.windows.WindowT = None, partitions: int = None, help: str = None, **kwargs: Any*) → faust.types.tables.TableT

Return type *TableT*[~KT, ~VT]

abstract SetTable(*name: str, *, window: faust.types.windows.WindowT = None, partitions: int = None, start_manager: bool = False, help: str = None, **kwargs: Any*) → faust.types.tables.TableT

Return type *TableT*[~KT, ~VT]

abstract SetGlobalTable(*name: str, *, window: faust.types.windows.WindowT = None, partitions: int = None, start_manager: bool = False, help: str = None, **kwargs: Any*) → faust.types.tables.TableT

Return type *TableT*[~KT, ~VT]

abstract page(*path: str, *, base: Type[faust.types.web.View] = <class 'faust.types.web.View'>, cors_options: Mapping[str, faust.types.web.ResourceOptions] = None, name: str = None*) → Callable[Union[Type[faust.types.web.View], Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Type[faust.types.web.View]]

Return type Callable[[Union[Type[*View*], Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]], Type[*View*]]

abstract table_route(*table: faust.types.tables.CollectionT*, *shard_param: str = None*, *, *query_param: str = None*, *match_info: str = None*, *exact_key: str = None*) → Callable[Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]]

Return type Callable[[Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]], Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]]

abstract command(**options: Any*, *base: Type[faust.types.app._AppCommand] = None*, ***kwargs: Any*) → Callable[Callable, Type[faust.types.app._AppCommand]]

Return type Callable[[Callable], Type[_AppCommand]]

abstract async start_client() → None

Return type None

abstract async maybe_start_client() → None

Return type None

abstract async send(*channel: Union[faust.types.channels.ChannelT, str]*, *key: Union[bytes, faust.types.core._ModelT, Any, None] = None*, *value: Union[bytes, faust.types.core._ModelT, Any] = None*, *partition: int = None*, *timestamp: float = None*, *headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None*, *schema: faust.types.app._SchemaT = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract LiveCheck(***kwargs: Any*) → faust.types.app._LiveCheck

Return type _LiveCheck

maybe_start_producer

Return type *ProducerT*[]

abstract is_leader() → bool

Return type bool

abstract FlowControlQueue(*maxsize: int = None*, ***, *clear_on_resume: bool = False*, *loop: asyncio.events.AbstractEventLoop = None*) → mode.utils.queues.ThrowableQueue

Return type ThrowableQueue

abstract Worker(***kwargs: Any*) → faust.types.app._Worker

Return type _Worker

abstract on_webserver_init(*web: faust.types.web.Web*) → None

Return type None

abstract on_rebalance_start() → None

Return type None

abstract on_rebalance_end() → None

Return type None

property conf

Return type _Settings

abstract property transport

Return type *TransportT*

abstract property producer_transport

Return type *TransportT*

abstract property cache

Return type *CacheBackendT*[]

abstract property producer

Return type *ProducerT*[]

abstract property consumer

Return type *ConsumerT*[]

tables

topics

abstract property monitor

Return type _Monitor

flow_control

abstract property http_client

Return type ClientSession

abstract property assignor

Return type *PartitionAssignorT*

abstract property router

Return type *RouterT*

abstract property serializers

Return type *RegistryT*

abstract property web

Return type *Web*

abstract property in_transaction

Return type bool

faust.types.assignor

faust.types.assignor.**TopicToPartitionMap**
alias of typing.MutableMapping

faust.types.assignor.**HostToPartitionMap**
alias of typing.MutableMapping

class faust.types.assignor.**PartitionAssignorT**(*app: faust.types.assignor._AppT*, *replicas: int = 0*) → None

abstract group_for_topic(*topic: str*) → int

Return type int

abstract assigned_standbys() → Set[faust.types.tuples.TP]

Return type Set[*TP*]

abstract assigned_actives() → Set[faust.types.tuples.TP]

Return type Set[*TP*]

abstract is_active(*tp: faust.types.tuples.TP*) → bool

Return type bool

abstract is_standby(*tp: faust.types.tuples.TP*) → bool

Return type bool

abstract key_store(*topic: str*, *key: bytes*) → yarl.URL

Return type URL

abstract table_metadata(*topic: str*) → MutableMapping[str, MutableMapping[str, List[int]]]

Return type MutableMapping[str, MutableMapping[str, List[int]]]

abstract tables_metadata() → MutableMapping[str, MutableMapping[str, List[int]]]

Return type MutableMapping[str, MutableMapping[str, List[int]]]

class faust.types.assignor.**LeaderAssignorT**(**, beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None

abstract is_leader() → bool

Return type bool

faust.types.auth

class faust.types.auth.**AuthProtocol**
An enumeration.

SSL = 'SSL'

PLAINTEXT = 'PLAINTEXT'

SASL_PLAINTEXT = 'SASL_PLAINTEXT'

SASL_SSL = 'SASL_SSL'

class faust.types.auth.**SASLMechanism**
An enumeration.

PLAIN = 'PLAIN'

GSSAPI = 'GSSAPI'

class faust.types.auth.**CredentialsT**(**args*, ***kwargs*)

faust.types.auth.**to_credentials**(*obj: Union[faust.types.auth.CredentialsT, ssl.SSLContext] = None*) → Optional[faust.types.auth.CredentialsT]

Return type Optional[*CredentialsT*]

faust.types.channels

class faust.types.channels.**ChannelT**(*app: faust.types.channels._AppT*, ***, *schema: faust.types.channels._SchemaT = None*, *key_type: faust.types.channels._ModelArg = None*, *value_type: faust.types.channels._ModelArg = None*, *is_iterator: bool = False*, *queue: mode.utils.queues.ThrowableQueue = None*, *maxsize: int = None*, *root: Optional[faust.types.channels.ChannelT] = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None

abstract clone(***, *is_iterator: bool = None*, ***kwargs: Any*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract clone_using_queue(*queue: asyncio.queues.Queue*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract stream(***kwargs: Any*) → faust.types.channels._StreamT

Return type _StreamT

abstract get_topic_name() → str

Return type str

abstract async send(*, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.channels._SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract send_soon(*, key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.channels._SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False, eager_partitioning: bool = False) → faust.types.tuples.FutureMessage

Return type *FutureMessage*[]

abstract as_future_message(key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.channels._SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, eager_partitioning: bool = False) → faust.types.tuples.FutureMessage

Return type *FutureMessage*[]

abstract async publish_message(*fut: faust.types.tuples.FutureMessage*, *wait: bool = True*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

maybe_declare

Return type None

abstract async declare() → None

Return type None

abstract prepare_key(*key: Union[bytes, faust.types.core._ModelT, Any, None]*, *key_serializer: Union[faust.types.codecs.CodecT, str, None]*, *schema: faust.types.channels._SchemaT = None*) → Any

Return type Any

abstract prepare_value(*value: Union[bytes, faust.types.core._ModelT, Any]*, *value_serializer: Union[faust.types.codecs.CodecT, str, None]*, *schema: faust.types.channels._SchemaT = None*) → Any

Return type Any

abstract async decode(*message: faust.types.tuples.Message*, *, *propagate: bool = False*) → faust.types.channels._EventT

Return type `_EventT`

abstract async deliver(*message: faust.types.tuples.Message*) → None

Return type `None`

abstract async put(*value: Any*) → None

Return type `None`

abstract async get(*, *timeout: Union[datetime.timedelta, float, str] = None*) → Any

Return type `Any`

abstract empty() → bool

Return type `bool`

abstract async on_key_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None

Return type `None`

abstract async on_value_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None

Return type `None`

abstract async on_decode_error(*exc: Exception*, *message: faust.types.tuples.Message*) → None

Return type `None`

abstract on_stop_iteration() → None

Return type `None`

abstract async throw(*exc: BaseException*) → None

Return type `None`

abstract derive(***kwargs: Any*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract property subscriber_count

Return type `int`

abstract property queue

Return type `ThrowableQueue`

faust.types.codecs

class faust.types.codecs.**CodecT**(*children: Tuple[CodecT, ...] = None*, ***kwargs: Any*)

Abstract type for an encoder/decoder.

See also:

faust.serializers.codecs.Codec.

abstract dumps(*obj: Any*) → bytes

Return type `bytes`

abstract loads(*s: bytes*) → Any

Return type Any

abstract clone(**children: faust.types.codecs.CodecT*) → faust.types.codecs.CodecT

Return type *CodecT*

faust.types.core

faust.types.core.**K = typing.Union[bytes, faust.types.core._ModelT, typing.Any, NoneType]**
Shorthand for the type of a key

faust.types.core.**V = typing.Union[bytes, faust.types.core._ModelT, typing.Any]**
Shorthand for the type of a value

faust.types.enums

class faust.types.enums.**ProcessingGuarantee**
An enumeration.

AT_LEAST_ONCE = 'at_least_once'

EXACTLY_ONCE = 'exactly_once'

faust.types.events

class faust.types.events.**EventT**(*app: faust.types.events._AppT, key: Union[bytes, faust.types.core._ModelT, Any, None], value: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], message: faust.types.tuples.Message*) → None

app

key

value

headers

message

acked

abstract async send(*channel: Union[str, faust.types.events._ChannelT], key: Union[bytes, faust.types.core._ModelT, Any, None] = None, value: Union[bytes, faust.types.core._ModelT, Any] = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.events._SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract async forward(*channel: Union[str, faust.types.events._ChannelT], key: Any = None, value: Any = None, partition: int = None, timestamp: float = None, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None] = None, schema: faust.types.events._SchemaT = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, callback: Callable[faust.types.tuples.FutureMessage, Union[None, Awaitable[None]]] = None, force: bool = False*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract ack() → bool

Return type bool

faust.types.fixups

class faust.types.fixups.**FixupT**(*app: faust.types.fixups._AppT*) → None

abstract enabled() → bool

Return type bool

abstract autodiscover_modules() → Iterable[str]

Return type Iterable[str]

abstract on_worker_init() → None

Return type None

faust.types.joins

class faust.types.joins.**JoinT**(**, stream: faust.types.streams.JoinableT, fields: Tuple[faust.types.models.FieldDescriptorT, ...]*) → None

abstract async process(*event: faust.types.events.EventT*) → Optional[faust.types.events.EventT]

Return type Optional[*EventT*[]]

faust.types.models

faust.types.models.**FieldMap**
alias of typing.Mapping

faust.types.models.**CoercionHandler**
alias of typing.Callable

class faust.types.models.**TypeCoerce**(**args*, ***kwargs*)

property target
Alias for field number 0

property handler
Alias for field number 1

class faust.types.models.**TypeInfo**(**args*, ***kwargs*)

property generic_type
Alias for field number 0

property member_type
Alias for field number 1

class faust.types.models.**ModelOptions**(**args*, ***kwargs*)

serializer = None

include_metadata = True

polymorphic_fields = False

allow_blessed_key = False

isodates = False

decimals = False

validation = False

coerce = False

coercions = None

date_parser = None

fields = None
Flattened view of __annotations__ in MRO order.

Type Index

fieldset = None
Set of required field names, for fast argument checking.

Type Index

descriptors = None
Mapping of field name to field descriptor.

Type Index

fieldpos = None
Positional argument index to field name. Used by Record.__init__ to map positional arguments to fields.

Type Index

optionalset = None
Set of optional field names, for fast argument checking.

Type Index

models = None
Mapping of fields that are ModelT

Type Index

modelattrs = None

field_coerce = None
Mapping of fields that need to be coerced. Key is the name of the field, value is the coercion handler function.

Type Index

defaults = None
Mapping of field names to default value.

initfield = None
Mapping of init field conversion callbacks.

polyindex = None
Index of field to polymorphic type

clone_defaults() → faust.types.models.ModelOptions

Return type *ModelOptions*

class faust.types.models.**ModelT**(**args: Any*, ***kwargs: Any*) → None

abstract classmethod from_data(*data: Any*, ***, *preferred_type: Type[ModelT] = None*) → faust.types.models.ModelT

Return type *ModelT*

abstract classmethod loads(*s: bytes*, ***, *default_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → faust.types.models.ModelT

Return type *ModelT*

abstract dumps(***, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → bytes

Return type bytes

abstract derive(**objects: faust.types.models.ModelT*, ***fields: Any*) → faust.types.models.ModelT

Return type *ModelT*

abstract to_representation() → Any

Return type Any

abstract is_valid() → bool

Return type bool

abstract validate() → List[faust.exceptions.ValidationError]

Return type List[ValidationError]

abstract validate_or_raise() → None

Return type None

abstract property validation_errors

Return type List[ValidationError]

class faust.types.models.**FieldDescriptorT**(***, *field: str = None*, *input_name: str = None*, *output_name: str = None*, *type: Type[T] = None*, *model: Type[faust.types.models.ModelT] = None*, *required: bool = True*, *default: T = None*, *parent: Optional[faust.types.models.FieldDescriptorT] = None*, *generic_type: Type = None*, *member_type: Type = None*, *exclude: bool = None*, *date_parser: Callable[Any, datetime.datetime] = None*, ***kwargs: Any*) → None

required = True

default = None

abstract clone(***kwargs: Any*) → faust.types.models.FieldDescriptorT

Return type *FieldDescriptorT*[~T]

abstract as_dict() → Mapping[str, Any]

Return type Mapping[str, Any]

abstract validate_all(*value: Any*) → Iterable[faust.exceptions.ValidationError]

Return type Iterable[ValidationError]

abstract validate(*value: T*) → Iterable[faust.exceptions.ValidationError]

Return type Iterable[ValidationError]

abstract prepare_value(*value: Any*) → Optional[T]

Return type Optional[~T]

abstract should_coerce(*value: Any*) → bool

Return type bool

abstract getattr(*obj: faust.types.models.ModelT*) → T

Return type ~T

abstract validation_error(*reason: str*) → faust.exceptions.ValidationError

Return type ValidationError

abstract property ident

Return type str

faust.types.router

Types for module faust.router.

class faust.types.router.**RouterT**(*app: faust.types.router._AppT*) → None
Router type class.

abstract key_store(*table_name: str, key: Union[bytes, faust.types.core._ModelT, Any, None]*) → yarl.URL

Return type URL

abstract table_metadata(*table_name: str*) → MutableMapping[str, MutableMapping[str, List[int]]]

Return type MutableMapping[str, MutableMapping[str, List[int]]]

abstract tables_metadata() → MutableMapping[str, MutableMapping[str, List[int]]]

Return type MutableMapping[str, MutableMapping[str, List[int]]]

abstract async route_req(*table_name: str, key: Union[bytes, faust.types.core._ModelT, Any, None], web: faust.types.web.Web, request: faust.types.web.Request*) → faust.types.web.Response

Return type *Response*

faust.types.sensors

class faust.types.sensors.**SensorInterfaceT**

abstract on_message_in(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None

Return type None

abstract on_stream_event_in(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*) → Optional[Dict]

Return type Optional[Dict[~KT, ~VT]]

abstract on_stream_event_out(*tp: faust.types.tuples.TP*, *offset: int*, *stream: faust.types.streams.StreamT*, *event: faust.types.events.EventT*, *state: Dict = None*) → None

Return type None

abstract on_topic_buffer_full(*topic: faust.types.topics.TopicT*) → None

Return type None

abstract on_message_out(*tp: faust.types.tuples.TP*, *offset: int*, *message: faust.types.tuples.Message*) → None

Return type None

abstract on_table_get(*table: faust.types.tables.CollectionT*, *key: Any*) → None

Return type None

abstract on_table_set(*table: faust.types.tables.CollectionT*, *key: Any*, *value: Any*) → None

Return type None

abstract on_table_del(*table: faust.types.tables.CollectionT*, *key: Any*) → None

Return type None

abstract on_commit_initiated(*consumer: faust.types.transports.ConsumerT*) → Any

Return type Any

abstract on_commit_completed(*consumer: faust.types.transports.ConsumerT*, *state: Any*) → None

Return type None

abstract on_send_initiated(*producer: faust.types.transports.ProducerT*, *topic: str*, *message: faust.types.tuples.PendingMessage*, *keysize: int*, *valsize: int*) → Any

Return type Any

abstract on_send_completed(*producer: faust.types.transports.ProducerT*, *state: Any*, *metadata: faust.types.tuples.RecordMetadata*) → None

Return type None

abstract on_send_error(*producer: faust.types.transports.ProducerT*, *exc: BaseException*, *state: Any*) → None

Return type None

abstract on_assignment_start(*assignor: faust.types.assignor.PartitionAssignorT*) → Dict

Return type Dict[~KT, ~VT]

abstract on_assignment_error(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*, *exc: BaseException*) → None

Return type None

abstract on_assignment_completed(*assignor: faust.types.assignor.PartitionAssignorT*, *state: Dict*) → None

Return type None

abstract on_rebalance_start(*app: faust.types.sensors._AppT*) → Dict

Return type Dict[~KT, ~VT]

abstract on_rebalance_return(*app: faust.types.sensors._AppT*, *state: Dict*) → None

Return type None

abstract on_rebalance_end(*app: faust.types.sensors._AppT*, *state: Dict*) → None

Return type None

abstract on_web_request_start(*app: faust.types.sensors._AppT*, *request: faust.types.web.Request*, ***, *view: faust.types.web.View = None*) → Dict

Return type Dict[~KT, ~VT]

abstract on_web_request_end(*app: faust.types.sensors._AppT, request: faust.types.web.Request, response: Optional[faust.types.web.Response], state: Dict, *, view: faust.types.web.View = None*) → None

Return type None

class faust.types.sensors.**SensorT**(***, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*) → None

class faust.types.sensors.**SensorDelegateT**

abstract add(*sensor: faust.types.sensors.SensorT*) → None

Return type None

abstract remove(*sensor: faust.types.sensors.SensorT*) → None

Return type None

faust.types.serializers

class faust.types.serializers.**RegistryT**(*key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = 'json'*) → None

abstract loads_key(*typ: Optional[faust.types.serializers._ModelArg], key: Optional[bytes], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → Union[bytes, faust.types.core._ModelT, Any, None]

Return type Union[bytes, _ModelT, Any, None]

abstract loads_value(*typ: Optional[faust.types.serializers._ModelArg], value: Optional[bytes], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → Any

Return type Any

abstract dumps_key(*typ: Optional[faust.types.serializers._ModelArg], key: Union[bytes, faust.types.core._ModelT, Any, None], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → Optional[bytes]

Return type Optional[bytes]

abstract dumps_value(*typ: Optional[faust.types.serializers._ModelArg], value: Union[bytes, faust.types.core._ModelT, Any], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → Optional[bytes]

Return type Optional[bytes]

class faust.types.serializers.**SchemaT**(*, *key_type: faust.types.serializers._ModelArg = None*, *value_type: faust.types.serializers._ModelArg = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *allow_empty: bool = None*) → None

key_type = None

value_type = None

key_serializer = None

value_serializer = None

allow_empty = False

abstract update(*, *key_type: faust.types.serializers._ModelArg = None*, *value_type: faust.types.serializers._ModelArg = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *allow_empty: bool = None*) → None

Return type None

abstract loads_key(*app: faust.types.serializers._AppT*, *message: faust.types.serializers._Message*, *, *loads: Callable = None*, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → KT

Return type ~KT

abstract loads_value(*app: faust.types.serializers._AppT*, *message: faust.types.serializers._Message*, *, *loads: Callable = None*, *serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → VT

Return type ~VT

abstract dumps_key(*app: faust.types.serializers._AppT, key: Union[bytes, faust.types.core._ModelT, Any, None], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]*) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

abstract dumps_value(*app: faust.types.serializers._AppT, value: Union[bytes, faust.types.core._ModelT, Any], *, serializer: Union[faust.types.codecs.CodecT, str, None] = None, headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]*) → Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

Return type Tuple[Any, Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]]

abstract on_dumps_key_prepare_headers(*key: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]*) → Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

Return type Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

abstract on_dumps_value_prepare_headers(*value: Union[bytes, faust.types.core._ModelT, Any], headers: Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]*) → Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

Return type Union[List[Tuple[str, bytes]], MutableMapping[str, bytes], None]

faust.types.settings

class faust.types.settings.**Settings**(*id: str, *, debug: bool = None, version: int = None, broker: Union[str, yarl.URL, List[yarl.URL]] = None, broker_client_id: str = None, broker_request_timeout: Union[datetime.timedelta, float, str] = None, broker_credentials: Union[faust.types.auth.CredentialsT, ssl.SSLContext] = None, broker_commit_every: int = None, broker_commit_interval: Union[datetime.timedelta, float, str] = None, broker_commit_livelock_soft_timeout: Union[datetime.timedelta, float, str] = None, broker_session_timeout: Union[datetime.timedelta, float, str] = None, broker_heartbeat_interval: Union[datetime.timedelta, float, str] = None, broker_check_crcs: bool = None, broker_max_poll_records: int = None, broker_max_poll_interval: int = None, broker_consumer: Union[str, yarl.URL, List[yarl.URL]] = None, broker_producer: Union[str, yarl.URL, List[yarl.URL]] = None, agent_supervisor: Union[_T, str] = None, store: Union[str, yarl.URL] = None, cache: Union[str, yarl.URL] = None, web: Union[str, yarl.URL] = None, web_enabled: bool = True, processing_guarantee: Union[str, faust.types.enums.ProcessingGuarantee] = None, timezone: datetime.tzinfo = None, autodiscover: Union[bool, Iterable[str], Callable[Iterable[str]]] = None, origin: str = None, canonical_url: Union[str, yarl.URL] = None, datadir: Union[pathlib.Path, str] = None, tabledir: Union[pathlib.Path, str] = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, logging_config: Dict = None, loghandlers: List[logging.Handler] = None, table_cleanup_interval: Union[datetime.timedelta, float, str] = None, table_standby_replicas: int = None, table_key_index_size: int = None, topic_replication_factor: int = None, topic_partitions: int = None, topic_allow_declare: bool = None, topic_disable_leader: bool = None, id_format: str = None, reply_to: str = None, reply_to_prefix: str = None, reply_create_topic: bool = None, reply_expires: Union[datetime.timedelta, float, str] = None, ssl_context: ssl.SSLContext = None, stream_buffer_maxsize: int = None, stream_wait_empty: bool = None, stream_ack_cancelled_tasks: bool = None, stream_ack_exceptions: bool = None, stream_publish_on_commit: bool = None, stream_recovery_delay: Union[datetime.timedelta, float, str] = None, producer_linger_ms: int = None, producer_max_batch_size: int = None, producer_acks: int = None, producer_max_request_size: int = None, producer_compression_type: str = None, producer_partitioner: Union[_T, str] = None, producer_request_timeout: Union[datetime.timedelta, float, str] = None, producer_api_version: str = None, consumer_max_fetch_size: int = None, consumer_auto_offset_reset: str = None, web_bind: str = None, web_port: int = None, web_host: str = None, web_transport: Union[str, yarl.URL] = None, web_in_thread: bool = None, web_cors_options: Mapping[str, faust.types.web.ResourceOptions] = None, worker_redirect_stdouts: bool = None, worker_redirect_stdouts_level: Union[int, str] = None,*

classmethod setting_names() → Set[str]

Return type Set[str]

```
id_format = '{id}-v{self.version}'
debug = False
ssl_context = None
autodiscover = False
broker_client_id = 'faust-1.9.0'
timezone = datetime.timezone.utc
broker_commit_every = 10000
broker_check_crcs = True
broker_max_poll_interval = 1000.0
key_serializer = 'raw'
value_serializer = 'json'
table_standby_replicas = 1
table_key_index_size = 1000
topic_replication_factor = 1
topic_partitions = 8
topic_allow_declare = True
topic_disable_leader = False
reply_create_topic = False
logging_config = None
stream_buffer_maxsize = 4096
stream_wait_empty = True
stream_ack_cancelled_tasks = True
stream_ack_exceptions = True
stream_publish_on_commit = False
producer_linger_ms = 0
producer_max_batch_size = 16384
producer_acks = -1
producer_max_request_size = 1000000
producer_compression_type = None
producer_api_version = 'auto'
consumer_max_fetch_size = 4194304
consumer_auto_offset_reset = 'earliest'
web_bind = '0.0.0.0'
web_port = 6066
```

web_host = 'build-10233069-project-230058-faust'

web_in_thread = False

web_cors_options = None

worker_redirect_stdouts = True

worker_redirect_stdouts_level = 'WARN'

reply_to_prefix = 'f-reply-'

property name

Return type str

property id

Return type str

property origin

Return type Optional[str]

property version

Return type int

property broker

Return type List[URL]

property broker_consumer

Return type List[URL]

property broker_producer

Return type List[URL]

property store

Return type URL

property web

Return type URL

property cache

Return type URL

property canonical_url

Return type URL

property datadir

Return type Path

property appdir

Return type Path

find_old_versiondirs() → Iterable[pathlib.Path]

Return type Iterable[Path]

property tabledir

Return type Path

property processing_guarantee

Return type *ProcessingGuarantee*

property broker_credentials

Return type Optional[*CredentialsT*]

property broker_request_timeout

Return type float

property broker_session_timeout

Return type float

property broker_heartbeat_interval

Return type float

property broker_commit_interval

Return type float

property broker_commit_livelock_soft_timeout

Return type float

property broker_max_poll_records

Return type Optional[int]

property producer_partitioner

Return type Optional[Callable[[Optional[bytes], Sequence[int], Sequence[int]], int]]

property producer_request_timeout

Return type float

property table_cleanup_interval

Return type float

property reply_expires

Return type float

property stream_recovery_delay

Return type float

property agent_supervisor

Return type Type[SupervisorStrategyT]

property web_transport

Return type URL

property Agent

Return type Type[*AgentT*[]]

property ConsumerScheduler

Return type Type[SchedulingStrategyT]

property Event

Return type Type[*EventT*[]]

property Schema

Return type Type[*SchemaT*[~KT, ~VT]]

property Stream

Return type Type[*StreamT*[+T_co]]

property Table

Return type Type[*TableT*[~KT, ~VT]]

property SetTable

Return type Type[*TableT*[~KT, ~VT]]

property GlobalTable

Return type Type[*GlobalTableT*[]]

property SetGlobalTable

Return type Type[*GlobalTableT*[]]

property TableManager

Return type Type[*TableManagerT*[]]

property Serializers

Return type Type[*RegistryT*]

property Worker

Return type Type[_WorkerT]

property PartitionAssignor

Return type Type[*PartitionAssignorT*]

property LeaderAssignor

Return type Type[*LeaderAssignorT*[]]

property Router

Return type Type[*RouterT*]

property Topic

Return type Type[*TopicT*[]]

property HttpClient

Return type Type[ClientSession]

property Monitor

Return type Type[*SensorT*[]]

faust.types.stores

class faust.types.stores.**StoreT**(*url: Union[str, yarl.URL], app: faust.types.stores._AppT, table: faust.types.stores._CollectionT, *, table_name: str = '', key_type: faust.types.stores._ModelArg = None, value_type: faust.types.stores._ModelArg = None, key_serializer: Union[faust.types.codecs.CodecT, str, None] = '', value_serializer: Union[faust.types.codecs.CodecT, str, None] = '', options: Mapping[str, Any] = None, **kwargs: Any*) → None

abstract persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]

Return type Optional[int]

abstract set_persisted_offset(*tp: faust.types.tuples.TP*, *offset: int*) → None

Return type None

abstract async need_active_standby_for(*tp: faust.types.tuples.TP*) → bool

Return type bool

abstract apply_changelog_batch(*batch: Iterable[faust.types.events.EventT], to_key: Callable[Any, KT], to_value: Callable[Any, VT]*) → None

Return type None

abstract reset_state() → None

Return type None

abstract async on_rebalance(*table: faust.types.stores._CollectionT, assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async on_recovery_completed(*active_tps: Set[faust.types.tuples.TP], standby_tps: Set[faust.types.tuples.TP]*) → None

Return type None

faust.types.streams

faust.types.streams.**Processor**
alias of typing.Callable

faust.types.streams.**GroupByKeyArg = typing.Union[faust.types.models.FieldDescriptorT, typi**
Type of the *key* argument to *Stream.group_by()*

class faust.types.streams.**StreamT**(*channel: AsyncIterator[T_co] = None, *, app: faust.types.streams._AppT = None, processors: Iterable[Callable[T]] = None, combined: List[faust.types.streams.JoinableT] = None, on_start: Callable = None, join_strategy: faust.types.streams._JoinT = None, beacon: mode.utils.types.trees.NodeT = None, concurrency_index: int = None, prev: Optional[faust.types.streams.StreamT] = None, active_partitions: Set[faust.types.tuples.TP] = None, enable_acks: bool = True, prefix: str = '', loop: asyncio.events.AbstractEventLoop = None*) → None

outbox = None

join_strategy = None

task_owner = None

current_event = None

active_partitions = None

concurrency_index = None

enable_acks = True

prefix = ''

abstract get_active_stream() → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract add_processor(*processor: Callable[T]*) → None

Return type None

abstract info() → Mapping[str, Any]

Return type Mapping[str, Any]

abstract clone(***kwargs: Any*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract async items() → AsyncIterator[Tuple[Union[bytes, faust.types.core._ModelT, Any, None], T_co]]

abstract async events() → AsyncIterable[faust.types.events.EventT]

abstract async take(*max_: int, within: Union[datetime.timedelta, float, str]*) → AsyncIterable[Sequence[T_co]]

abstract enumerate(*start: int = 0*) → AsyncIterable[Tuple[int, T_co]]

Return type AsyncIterable[Tuple[int, +T_co]]

abstract through(*channel: Union[str, faust.types.channels.ChannelT]*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract echo(**channels: Union[str, faust.types.channels.ChannelT]*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract group_by(*key: Union[faust.types.models.FieldDescriptorT, Callable[T, Union[bytes, faust.types.core._ModelT, Any, None]]], *, name: str = None, topic: faust.types.topics.TopicT = None*) → faust.types.streams.StreamT

Return type *StreamT*[+T_co]

abstract derive_topic(*name: str, *, schema: faust.types.streams._SchemaT = None, key_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, value_type: Union[Type[faust.types.models.ModelT], Type[bytes], Type[str]] = None, prefix: str = '', suffix: str = ''*) → faust.types.topics.TopicT

Return type *TopicT*[]

abstract async throw(*exc: BaseException*) → None

Return type None

abstract async send(*value: T_contra*) → None

Return type None

abstract async ack(*event: faust.types.events.EventT*) → bool

Return type bool

faust.types.tables

faust.types.tables.**RecoverCallback**
alias of typing.Callable

faust.types.tables.**ChangelogEventCallback**
alias of typing.Callable

faust.types.tables.**WindowCloseCallback**
alias of typing.Callable

faust.types.tables.**CollectionTps**
alias of typing.MutableMapping

class faust.types.tables.**CollectionT**(*app: faust.types.tables._AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.tables._SchemaT = None, key_type: faust.types.tables._ModelArg = None, value_type: faust.types.tables._ModelArg = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

abstract property changelog_topic

Return type *TopicT*[]

abstract apply_changelog_batch(*batch: Iterable[faust.types.events.EventT]*) → None

Return type None

abstract persisted_offset(*tp: faust.types.tuples.TP*) → Optional[int]

Return type Optional[int]

abstract async need_active_standby_for(*tp: faust.types.tuples.TP*) → bool

Return type bool

abstract reset_state() → None

Return type None

abstract send_changelog(*partition: Optional[int], key: Any, value: Any, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*) → faust.types.tuples.FutureMessage

Return type *FutureMessage*[]

abstract partition_for_key(*key: Any*) → Optional[int]

Return type Optional[int]

abstract on_window_close(*key: Any*, *value: Any*) → None

Return type None

abstract async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async on_changelog_event(*event: faust.types.events.EventT*) → None

Return type None

abstract on_recover(*fun: Callable[Awaitable[None]]*) → Callable[Awaitable[None]]

Return type Callable[[], Awaitable[None]]

abstract async on_recovery_completed(*active_tps: Set[faust.types.tuples.TP], standby_tps: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async call_recover_callbacks() → None

Return type None

class faust.types.tables.**TableT**(*app: faust.types.tables._AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.tables._SchemaT = None, key_type: faust.types.tables._ModelArg = None, value_type: faust.types.tables._ModelArg = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

abstract using_window(*window: faust.types.windows.WindowT, *, key_index: bool = False*) → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract hopping(*size: Union[datetime.timedelta, float, str], step: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract tumbling(*size: Union[datetime.timedelta, float, str], expires: Union[datetime.timedelta, float, str] = None, key_index: bool = False*) → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract as_ansitable(***kwargs: Any*) → str

Return type str

class faust.types.tables.**GlobalTableT**(*app: faust.types.tables._AppT, *, name: str = None, default: Callable[Any] = None, store: Union[str, yarl.URL] = None, schema: faust.types.tables._SchemaT = None, key_type: faust.types.tables._ModelArg = None, value_type: faust.types.tables._ModelArg = None, partitions: int = None, window: faust.types.windows.WindowT = None, changelog_topic: faust.types.topics.TopicT = None, help: str = None, on_recover: Callable[Awaitable[None]] = None, on_changelog_event: Callable[faust.types.events.EventT, Awaitable[None]] = None, recovery_buffer_size: int = 1000, standby_buffer_size: int = None, extra_topic_configs: Mapping[str, Any] = None, options: Mapping[str, Any] = None, use_partitioner: bool = False, on_window_close: Callable[[Any, Any], None] = None, **kwargs: Any*) → None

class faust.types.tables.**TableManagerT**(*app: faust.types.tables._AppT, **kwargs: Any*) → None

abstract add(*table: faust.types.tables.CollectionT*) → faust.types.tables.CollectionT

Return type *CollectionT*[]

abstract persist_offset_on_commit(*store: faust.types.stores.StoreT, tp: faust.types.tuples.TP, offset: int*) → None

Return type None

abstract on_commit(*offsets: MutableMapping[faust.types.tuples.TP, int]*) → None

Return type None

abstract async on_rebalance(*assigned: Set[faust.types.tuples.TP], revoked: Set[faust.types.tuples.TP], newly_assigned: Set[faust.types.tuples.TP]*) → None

Return type None

abstract property changelog_topics

Return type Set[str]

class faust.types.tables.**WindowSetT**(*key: KT, table: faust.types.tables.TableT, wrapper: faust.types.tables.WindowWrapperT, event: faust.types.events.EventT = None*) → None

abstract apply(*op: Callable[[VT, VT], VT], value: VT, event: faust.types.events.EventT = None*) → faust.types.tables.WindowSetT

Return type *WindowSetT*[~KT, ~VT]

abstract value(*event: faust.types.events.EventT = None*) → VT

Return type ~VT

abstract current(*event: faust.types.events.EventT = None*) → VT

Return type ~VT

abstract now() → VT

Return type ~VT

abstract delta(*d: Union[datetime.timedelta, float, str], event: faust.types.events.EventT = None*) → VT

Return type ~VT

class faust.types.tables.**WindowedItemsViewT**(*mapping: faust.types.tables.WindowWrapperT, event: faust.types.events.EventT = None*)

abstract now() → Iterator[Tuple[Any, Any]]

Return type Iterator[Tuple[Any, Any]]

abstract current(*event: faust.types.events.EventT = None*) → Iterator[Tuple[Any, Any]]

Return type Iterator[Tuple[Any, Any]]

abstract delta(*d: Union[datetime.timedelta, float, str], event: faust.types.events.EventT = None*) → Iterator[Tuple[Any, Any]]

Return type Iterator[Tuple[Any, Any]]

class faust.types.tables.**WindowedValuesViewT**(*mapping: faust.types.tables.WindowWrapperT, event: faust.types.events.EventT = None*)

abstract now() → Iterator[Any]

Return type Iterator[Any]

abstract current(*event: faust.types.events.EventT = None*) → Iterator[Any]

Return type Iterator[Any]

abstract delta(*d: Union[datetime.timedelta, float, str], event: faust.types.events.EventT = None*) → Iterator[Any]

Return type Iterator[Any]

class faust.types.tables.**WindowWrapperT**(*table: faust.types.tables.TableT, *, relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None] = None, key_index: bool = False, key_index_table: faust.types.tables.TableT = None*) → None

abstract property name

Return type str

abstract clone(*relative_to: Union[faust.types.tables._FieldDescriptorT, Callable[Optional[faust.types.events.EventT], Union[float, datetime.datetime]], datetime.datetime, float, None]*) → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract relative_to_now() → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract relative_to_field(*field: faust.types.tables._FieldDescriptorT*) → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract relative_to_stream() → faust.types.tables.WindowWrapperT

Return type *WindowWrapperT*[]

abstract get_timestamp(*event: faust.types.events.EventT = None*) → float

Return type float

abstract keys() → KeysView

Return type KeysView[~KT]

abstract on_set_key(*key: Any*, *value: Any*) → None

Return type None

abstract on_del_key(*key: Any*) → None

Return type None

abstract as_ansitable(***kwargs: Any*) → str

Return type str

property get_relative_timestamp

Return type Optional[Callable[[Optional[*EventT*[]]], Union[float, datetime]]]

faust.types.topics

class faust.types.topics.**TopicT**(*app: faust.types.topics._AppT*, ***, *topics: Sequence[str] = None*, *pattern: Union[str, Pattern[~AnyStr]] = None*, *schema: faust.types.topics._SchemaT = None*, *key_type: faust.types.topics._ModelArg = None*, *value_type: faust.types.topics._ModelArg = None*, *is_iterator: bool = False*, *partitions: int = None*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *replicas: int = None*, *acks: bool = True*, *internal: bool = False*, *config: Mapping[str, Any] = None*, *queue: mode.utils.queues.ThrowableQueue = None*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *maxsize: int = None*, *root: faust.types.channels.ChannelT = None*, *active_partitions: Set[faust.types.tuples.TP] = None*, *allow_empty: bool = False*, *has_prefix: bool = False*, *loop: asyncio.events.AbstractEventLoop = None*) → None

topics = None
Iterable/Sequence of topic names to subscribe to.

retention = None
expiry time in seconds for messages in the topic.

Type Topic retention setting

compacting = None
Flag that when enabled means the topic can be "compacted": if the topic is a log of key/value pairs, the broker can delete old values for the same key.

replicas = None
Number of replicas for topic.

config = None
Additional configuration as a mapping.

acks = None
Enable acks for this topic.

internal = None
it's owned by us and we are allowed to create or delete the topic as necessary.

Type Mark topic as internal

has_prefix = False

abstract property pattern

Return type Optional[Pattern[AnyStr]]

abstract property partitions

Return type Optional[int]

abstract derive(***kwargs: Any*) → faust.types.channels.ChannelT

Return type *ChannelT*[]

abstract derive_topic(**, topics: Sequence[str] = None, schema: faust.types.topics._SchemaT = None, key_type: faust.types.topics._ModelArg = None, value_type: faust.types.topics._ModelArg = None, partitions: int = None, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, internal: bool = False, config: Mapping[str, Any] = None, prefix: str = '', suffix: str = '', **kwargs: Any*) → faust.types.topics.TopicT

Return type *TopicT*[]

faust.types.transports

faust.types.transports.**ConsumerCallback**
alias of typing.Callable

faust.types.transports.**TPorTopicSet**
alias of typing.AbstractSet

faust.types.transports.**PartitionsRevokedCallback**
alias of typing.Callable

faust.types.transports.**PartitionsAssignedCallback**
alias of typing.Callable

faust.types.transports.**PartitionerT**
alias of typing.Callable

class faust.types.transports.**ProducerT**(*transport: faust.types.transports.TransportT, loop: asyncio.events.AbstractEventLoop = None, **kwargs: Any*) → None

transport = None
The transport that created this Producer.

abstract async send(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → Awaitable[faust.types.tuples.RecordMetadata]

Return type Awaitable[*RecordMetadata*]

abstract async send_and_wait(*topic: str, key: Optional[bytes], value: Optional[bytes], partition: Optional[int], timestamp: Optional[float], headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], *, transactional_id: str = None*) → faust.types.tuples.RecordMetadata

Return type *RecordMetadata*

abstract async create_topic(*topic: str, partitions: int, replication: int, *, config: Mapping[str, Any] = None, timeout: Union[datetime.timedelta, float, str] = 1000.0, retention: Union[datetime.timedelta, float, str] = None, compacting: bool = None, deleting: bool = None, ensure_created: bool = False*) → None

Return type None

abstract key_partition(*topic: str, key: bytes*) → faust.types.tuples.TP

Return type *TP*

abstract async flush() → None

Return type None

abstract async begin_transaction(*transactional_id: str*) → None

Return type None

abstract async commit_transaction(*transactional_id: str*) → None

Return type None

abstract async abort_transaction(*transactional_id: str*) → None

Return type None

abstract async stop_transaction(*transactional_id: str*) → None

Return type None

abstract async maybe_begin_transaction(*transactional_id: str*) → None

Return type None

abstract async commit_transactions(*tid_to_offset_map: Mapping[str, Mapping[faust.types.tuples.TP, int]], group_id: str, start_new_transaction: bool = True*) → None

Return type None

abstract supports_headers() → bool

Return type bool

class faust.types.transports.**TransactionManagerT**(*transport: faust.types.transports.TransportT*, *loop: asyncio.events.AbstractEventLoop = None*, **, consumer: faust.types.transports.ConsumerT*, *producer: faust.types.transports.ProducerT*, ***kwargs: Any*) → None

abstract async on_partitions_revoked(*revoked: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async on_rebalance(*assigned: Set[faust.types.tuples.TP]*, *revoked: Set[faust.types.tuples.TP]*, *newly_assigned: Set[faust.types.tuples.TP]*) → None

Return type None

abstract async commit(*offsets: Mapping[faust.types.tuples.TP, int]*, *start_new_transaction: bool = True*) → bool

Return type bool

async begin_transaction(*transactional_id: str*) → None

Return type None

async commit_transaction(*transactional_id: str*) → None

Return type None

async abort_transaction(*transactional_id: str*) → None

Return type None

async stop_transaction(*transactional_id: str*) → None

Return type None

async maybe_begin_transaction(*transactional_id: str*) → None

Return type None

async commit_transactions(*tid_to_offset_map: Mapping[str, Mapping[faust.types.tuples.TP, int]]*, *group_id: str*, *start_new_transaction: bool = True*) → None

Return type None

class faust.types.transports.**ConsumerT**(*transport: faust.types.transports.TransportT*, *callback: Callable[faust.types.tuples.Message, Awaitable]*, *on_partitions_revoked: Callable[Set[faust.types.tuples.TP], Awaitable[None]]*, *on_partitions_assigned: Callable[Set[faust.types.tuples.TP], Awaitable[None]]*, **, commit_interval: float = None*, *loop: asyncio.events.AbstractEventLoop = None*, ***kwargs: Any*) → None

transport = None

The transport that created this Consumer.

commit_interval = None
How often we commit topic offsets. See *broker_commit_interval*.

randomly_assigned_topics = None
Set of topic names that are considered "randomly assigned". This means we don't crash if it's not part of our assignment. Used by e.g. the leader assignor service.

abstract async create_topic(*topic: str*, *partitions: int*, *replication: int*, ***, *config: Mapping[str, Any] = None*, *timeout: Union[datetime.timedelta, float, str] = 1000.0*, *retention: Union[datetime.timedelta, float, str] = None*, *compacting: bool = None*, *deleting: bool = None*, *ensure_created: bool = False*) → None

Return type None

abstract async subscribe(*topics: Iterable[str]*) → None

Return type None

abstract async getmany(*timeout: float*) → AsyncIterator[Tuple[faust.types.tuples.TP, faust.types.tuples.Message]]

abstract track_message(*message: faust.types.tuples.Message*) → None

Return type None

abstract async perform_seek() → None

Return type None

abstract ack(*message: faust.types.tuples.Message*) → bool

Return type bool

abstract async wait_empty() → None

Return type None

abstract assignment() → Set[faust.types.tuples.TP]

Return type Set[*TP*]

abstract highwater(*tp: faust.types.tuples.TP*) → int

Return type int

abstract stop_flow() → None

Return type None

abstract resume_flow() → None

Return type None

abstract pause_partitions(*tps: Iterable[faust.types.tuples.TP]*) → None

Return type None

abstract resume_partitions(*tps: Iterable[faust.types.tuples.TP]*) → None

Return type None

abstract async position(*tp: faust.types.tuples.TP*) → Optional[int]

Return type Optional[int]

abstract async seek(*partition: faust.types.tuples.TP*, *offset: int*) → None

Return type None

abstract async seek_wait(*partitions: Mapping[faust.types.tuples.TP, int]*) → None

Return type None

abstract async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]] = None, start_new_transaction: bool = True*) → bool

Return type bool

abstract async on_task_error(*exc: BaseException*) → None

Return type None

abstract async earliest_offsets(**partitions: faust.types.tuples.TP*) → Mapping[faust.types.tuples.TP, int]

Return type Mapping[*TP*, int]

abstract async highwaters(**partitions: faust.types.tuples.TP*) → Mapping[faust.types.tuples.TP, int]

Return type Mapping[*TP*, int]

abstract topic_partitions(*topic: str*) → Optional[int]

Return type Optional[int]

abstract key_partition(*topic: str, key: Optional[bytes], partition: int = None*) → Optional[int]

Return type Optional[int]

abstract close() → None

Return type None

abstract property unacked

Return type Set[*Message*]

class faust.types.transports.**ConductorT**(*app: faust.types.transports._AppT, **kwargs: Any*) → None

abstract acks_enabled_for(*topic: str*) → bool

Return type bool

abstract async commit(*topics: AbstractSet[Union[str, faust.types.tuples.TP]]*) → bool

Return type bool

abstract async wait_for_subscriptions() → None

Return type None

abstract async maybe_wait_for_subscriptions() → None

Return type None

abstract async on_partitions_assigned(*assigned: Set[faust.types.tuples.TP]*) → None

Return type None

class faust.types.transports.**TransportT**(*url: List[yarl.URL], app: faust.types.transports._AppT, loop: asyncio.events.AbstractEventLoop = None*) → None

Consumer = None

The Consumer class used for this type of transport.

Producer = None
The Producer class used for this type of transport.

TransactionManager = None
The TransactionManager class used for managing multiple transactions.

Conductor = None
The Conductor class used to delegate messages from Consumer to streams.

Fetcher = None
The Fetcher service used for this type of transport.

app = None
The *faust.App* that created this transport.

url = None
//localhost).

Type The URL to use for this transport (e.g. kafka

driver_version = None
String identifying the underlying driver used for this transport. E.g. for aiokafka this could be `aiokafka 0.4.1`.

abstract create_consumer(*callback: Callable[faust.types.tuples.Message, Awaitable], **kwargs: Any*) → faust.types.transports.ConsumerT

Return type *ConsumerT*[]

abstract create_producer(***kwargs: Any*) → faust.types.transports.ProducerT

Return type *ProducerT*[]

abstract create_transaction_manager(*consumer: faust.types.transports.ConsumerT*, *producer: faust.types.transports.ProducerT*, ***kwargs: Any*) → faust.types.transports.TransactionManagerT

Return type *TransactionManagerT*[]

abstract create_conductor(***kwargs: Any*) → faust.types.transports.ConductorT

Return type *ConductorT*[]

faust.types.tuples

class faust.types.tuples.**TP**(**args*, ***kwargs*)

property topic
Alias for field number 0

property partition
Alias for field number 1

class faust.types.tuples.**RecordMetadata**(**args*, ***kwargs*)

property topic
Alias for field number 0

property partition
Alias for field number 1

property topic_partition
Alias for field number 2

property offset
Alias for field number 3

property timestamp
Alias for field number 4

property timestamp_type
Alias for field number 5

class faust.types.tuples.**PendingMessage**(**args*, ***kwargs*)

property channel
Alias for field number 0

property key
Alias for field number 1

property value
Alias for field number 2

property partition
Alias for field number 3

property timestamp
Alias for field number 4

property headers
Alias for field number 5

property key_serializer
Alias for field number 6

property value_serializer
Alias for field number 7

property callback
Alias for field number 8

property topic
Alias for field number 9

property offset
Alias for field number 10

class faust.types.tuples.**FutureMessage**(*message: faust.types.tuples.PendingMessage*) → None

set_result(*result: faust.types.tuples.RecordMetadata*) → None
Mark the future done and set its result.

If the future is already done when this method is called, raises InvalidStateError.

Return type None

class faust.types.tuples.**Message**(*topic: str, partition: int, offset: int, timestamp: float, timestamp_type: int, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], key: Optional[bytes], value: Optional[bytes], checksum: Optional[bytes], serialized_key_size: int = None, serialized_value_size: int = None, tp: faust.types.tuples.TP = None, time_in: float = None, time_out: float = None, time_total: float = None*) → None

use_tracking = False

topic

partition

offset

timestamp

timestamp_type

headers

key

value

checksum

serialized_key_size

serialized_value_size

acked

refcount

tp

tracked

time_in
Monotonic timestamp of when the consumer received this message.

time_out
Monotonic timestamp of when the consumer acknowledged this message.

time_total
Total processing time (in seconds), or None if the event is still processing.

ack(*consumer: faust.types.tuples._ConsumerT, n: int = 1*) → bool

Return type bool

on_final_ack(*consumer: faust.types.tuples._ConsumerT*) → bool

Return type bool

incref(*n: int = 1*) → None

Return type None

decref(*n: int = 1*) → int

Return type int

classmethod from_message(*message: Any, tp: faust.types.tuples.TP*) → faust.types.tuples.Message

Return type *Message*

span

class faust.types.tuples.**ConsumerMessage**(*topic: str, partition: int, offset: int, timestamp: float, timestamp_type: int, headers: Union[List[Tuple[str, bytes]], Mapping[str, bytes], None], key: Optional[bytes], value: Optional[bytes], checksum: Optional[bytes], serialized_key_size: int = None, serialized_value_size: int = None, tp: faust.types.tuples.TP = None, time_in: float = None, time_out: float = None, time_total: float = None*) → None

Message type used by Kafka Consumer.

acked

checksum

headers

key

offset

partition

refcount

serialized_key_size

serialized_value_size

span

time_in

time_out

time_total

timestamp

timestamp_type

topic

tp

tracked

use_tracking = True

value

on_final_ack(*consumer: faust.types.tuples._ConsumerT*) → bool

Return type bool

faust.types.tuples.**tp_set_to_map**(*tps: Set[faust.types.tuples.TP]*) → MutableMapping[str, Set[faust.types.tuples.TP]]

Return type MutableMapping[str, Set[*TP*]]

faust.types.tuples.**MessageSentCallback**

alias of typing.Callable

faust.types.web

faust.types.web.**HttpClientT**
alias of aiohttp.client.ClientSession

class faust.types.web.**Request**

class faust.types.web.**Response**

class faust.types.web.**Web**

class faust.types.web.**View**

faust.types.web.**ViewHandlerMethod**
alias of typing.Callable

faust.types.web.**ViewDecorator**
alias of typing.Callable

class faust.types.web.**ResourceOptions**(**args*, ***kwargs*)
CORS Options for specific route, or defaults.

property allow_credentials
Alias for field number 0

property expose_headers
Alias for field number 1

property allow_headers
Alias for field number 2

property max_age
Alias for field number 3

property allow_methods
Alias for field number 4

class faust.types.web.**CacheBackendT**(*app: faust.types.web._AppT*, *url: Union[yarl.URL, str] = 'memory://'*, ***kwargs: Any*) → None

abstract async get(*key: str*) → Optional[bytes]

Return type Optional[bytes]

abstract async set(*key: str*, *value: bytes*, *timeout: float*) → None

Return type None

abstract async delete(*key: str*) → None

Return type None

class faust.types.web.**CacheT**(*timeout: Union[datetime.timedelta, float, str] = None, key_prefix: str = None, backend: Union[Type[faust.types.web.CacheBackendT], str] = None, **kwargs: Any*) → None

abstract view(*timeout: Union[datetime.timedelta, float, str] = None*, *include_headers: bool = False*, *key_prefix: str = None*, ***kwargs: Any*) → Callable[Callable, Callable]

Return type Callable[[Callable], Callable]

class faust.types.web.**BlueprintT**(**args*, ***kwargs*)

abstract cache(*timeout: Union[datetime.timedelta, float, str] = None, include_headers: bool = False, key_prefix: str = None, backend: Union[Type[faust.types.web.CacheBackendT], str] = None*) → faust.types.web.CacheT

Return type *CacheT*

abstract route(*uri: str, *, name: Optional[str] = None, base: Type[faust.types.web.View] = <class 'faust.types.web.View'>*) → Callable[Union[Type[faust.types.web.View], Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Union[Type[faust.types.web.View], Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]]

Return type Callable[[Union[Type[*View*], Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]], Union[Type[*View*], Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]]

abstract static(*uri: str, file_or_directory: Union[str, pathlib.Path], *, name: Optional[str] = None*) → None

Return type None

abstract register(*app: faust.types.web._AppT, *, url_prefix: Optional[str] = None*) → None

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

Return type None

abstract init_webserver(*web: faust.types.web.Web*) → None

Return type None

abstract on_webserver_init(*web: faust.types.web.Web*) → None

Return type None

faust.types.windows

Types related to windowing.

faust.types.windows.**WindowRange**

alias of typing.Tuple

class faust.types.windows.**WindowT**(**args*, ***kwargs*)

Type class for windows.

expires = None

tz = None

abstract ranges(*timestamp: float*) → List[Tuple[float, float]]

Return type List[Tuple[float, float]]

abstract stale(*timestamp: float*, *latest_timestamp: float*) → bool

Return type bool

abstract current(*timestamp: float*) → Tuple[float, float]

Return type Tuple[float, float]

abstract earliest(*timestamp: float*) → Tuple[float, float]

Return type Tuple[float, float]

abstract delta(*timestamp: float, d: Union[datetime.timedelta, float, str]*) → Tuple[float, float]

Return type Tuple[float, float]

1.6.14 Utils

faust.utils.codegen

Utilities for generating code at runtime.

faust.utils.codegen.**Function**(*name: str, args: List[str], body: List[str], *, globals: Dict[str, Any] = None, locals: Dict[str, Any] = None, return_type: Any = <object object>, argsep: str = ', '*) → Callable

Generate a function from Python.

Return type Callable

faust.utils.codegen.**Method**(*name: str, args: List[str], body: List[str], **kwargs: Any*) → Callable

Generate Python method.

Return type Callable

faust.utils.codegen.**InitMethod**(*args: List[str], body: List[str], **kwargs: Any*) → Callable[None]

Generate __init__ method.

Return type Callable[[], None]

faust.utils.codegen.**HashMethod**(*attrs: List[str], **kwargs: Any*) → Callable[None]

Generate __hash__ method.

Return type Callable[[], None]

faust.utils.codegen.**EqMethod**(*fields: List[str], **kwargs: Any*) → Callable[None]

Generate __eq__ method.

Return type Callable[[], None]

faust.utils.codegen.**NeMethod**(*fields: List[str], **kwargs: Any*) → Callable[None]

Generate __ne__ method.

Return type Callable[[], None]

faust.utils.codegen.**GeMethod**(*fields: List[str], **kwargs: Any*) → Callable[None]

Generate __ge__ method.

Return type Callable[[], None]

faust.utils.codegen.**GtMethod**(*fields: List[str], **kwargs: Any*) → Callable[None]

Generate __gt__ method.

Return type Callable[[], None]

faust.utils.codegen.**LeMethod**(*fields: List[str], **kwargs: Any*) → Callable[None]
Generate __le__ method.

Return type Callable[[], None]

faust.utils.codegen.**LtMethod**(*fields: List[str], **kwargs: Any*) → Callable[None]
Generate __lt__ method.

Return type Callable[[], None]

faust.utils.codegen.**CompareMethod**(*name: str, op: str, fields: List[str], **kwargs: Any*) → Callable[None]
Generate object comparison method.

Excellent for __eq__, __le__, etc.

Examples

The example:

```
CompareMethod(
    name='__eq__',
    op='==',
    fields=['x', 'y'],
)
```

Generates a method like this:

```
def __eq__(self, other):
   if other.__class__ is self.__class__:
        return (self.x,self.y) == (other.x,other.y)
    return NotImplemented
```

Return type Callable[[], None]

faust.utils.codegen.**reprkwargs**(*kwargs: Mapping[str, Any], *, sep: str = ', ', fmt: str = '{0}={1}'*) → str

Return type str

faust.utils.codegen.**reprcall**(*name: str*, *args: Tuple = ()*, *kwargs: Mapping[str, Any] = {}*, ***, *sep: str = ', '*) → str

Return type str

faust.utils.cron

Crontab Utilities.

faust.utils.cron.**secs_for_next**(*cron_format: str*, *tz: datetime.tzinfo = None*) → float
Return seconds until next execution given Crontab style format.

Return type float

faust.utils.functional

Functional utilities.

faust.utils.functional.**consecutive_numbers**(*it: Iterable[int]*) → Iterator[Sequence[int]]
Find runs of consecutive numbers.

Notes

See https://docs.python.org/2.6/library/itertools.html#examples

Return type Iterator[Sequence[int]]

faust.utils.functional.**deque_prune**(*l: Deque[T], max: int = None*) → Optional[T]
Prune oldest element in deque if size exceeds `max`.

Return type Optional[~T]

faust.utils.functional.**deque_pushpopmax**(*l: Deque[T], item: T, max: int = None*) → Optional[T]
Append to deque and remove oldest element if size exceeds `max`.

Return type Optional[~T]

faust.utils.iso8601

Parsing ISO-8601 string and converting to `datetime`.

faust.utils.iso8601.**parse**(*datetime_string: str*) → datetime.datetime
Parse and convert ISO 8601 string into a datetime object.

Return type datetime

faust.utils.json

JSON utilities.

faust.utils.json.**str_to_decimal**(*s: str*, *maxlen: int = 1000*) → Optional[decimal.Decimal]
Convert string to `Decimal`.

Parameters

- **s** (*str*) – Number to convert.
- **maxlen** (*int*) – Max length of string. Default is 100.

Raises **ValueError** – if length exceeds maximum length, or if value is not a valid number (e.g. Inf, NaN or sNaN).

Return type Optional[Decimal]

Returns Converted number.

Return type Decimal

class faust.utils.json.**JSONEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)
Faust customized `json.JSONEncoder`.

Our version supports additional types like `UUID`, and importantly includes microsecond information in datetimes.

default(*o: Any*, ***, *callback: Callable[Any, Any] = <function on_default>*) → Any
Try to convert non-built-in json type to json.

Return type Any

faust.utils.json.**dumps**(*obj: Any*, *json_dumps: Callable = <function dumps>*, *cls: Type[faust.utils.json.JSONEncoder] = <class 'faust.utils.json.JSONEncoder'>*, ***kwargs: Any*) → str
Serialize to json. See `json.dumps()`.

Return type str

faust.utils.json.**loads**(*s: str*, *json_loads: Callable = <function loads>*, ***kwargs: Any*) → Any
Deserialize json string. See `json.loads()`.

Return type Any

faust.utils.platforms

Platform/OS utilities.

faust.utils.platforms.**max_open_files**() → Optional[int]
Return max number of open files, or None.

Return type Optional[int]

faust.utils.tracing

OpenTracing utilities.

faust.utils.tracing.**current_span**() → Optional[opentracing.span.Span]
Get the current span for this context (if any).

Return type Optional[Span]

faust.utils.tracing.**set_current_span**(*span: opentracing.span.Span*) → None
Set the current span for the current context.

Return type None

faust.utils.tracing.**noop_span**() → opentracing.span.Span
Return a span that does nothing when traced.

Return type Span

faust.utils.tracing.**finish_span**(*span: Optional[opentracing.span.Span]*, ***, *error: BaseException = None*) → None
Finish span, and optionally set error tag.

Return type None

faust.utils.tracing.**operation_name_from_fun**(*fun: Any*) → str
Generate opentracing name from function.

Return type str

faust.utils.tracing.**traced_from_parent_span**(*parent_span: opentracing.span.Span = None*, *callback: Callable = None*, ***extra_context: Any*) → Callable
Decorate function to be traced from parent span.

Return type Callable

faust.utils.tracing.**call_with_trace**(*span: opentracing.span.Span, fun: Callable, callback: Optional[Tuple[Callable, Tuple[Any, ...]]], *args: Any, **kwargs: Any*) → Any

Call function and trace it from parent span.

Return type Any

faust.utils.urls

URL utilities - Working with URLs.

faust.utils.urls.**urllist**(*arg: Union[yarl.URL, str, List[str], List[yarl.URL]], *, default_scheme: str = None*) → List[yarl.URL]

Create list of URLs.

You can pass in a comma-separated string, or an actual list and this will convert that into a list of yarl.URL objects.

Return type List[URL]

faust.utils.venusian

Venusian (see venusian).

We define our own interface so we don't have to specify the callback argument.

class faust.utils.venusian.**Scanner**(***kw*)

scan(*package*, *categories=None*, *onerror=None*, *ignore=None*)

Scan a Python package and any of its subpackages. All top-level objects will be considered; those marked with venusian callback attributes related to category will be processed.

The package argument should be a reference to a Python package or module object.

The categories argument should be sequence of Venusian callback categories (each category usually a string) or the special value None which means all Venusian callback categories. The default is None.

The onerror argument should either be None or a callback function which behaves the same way as the onerror callback function described in http://docs.python.org/library/pkgutil.html#pkgutil.walk_packages . By default, during a scan, Venusian will propagate all errors that happen during its code importing process, including ImportError. If you use a custom onerror callback, you can change this behavior.

Here's an example onerror callback that ignores ImportError:

```
import sys
def onerror(name):
    if not issubclass(sys.exc_info()[0], ImportError):
        raise # reraise the last exception
```

The name passed to onerror is the module or package dotted name that could not be imported due to an exception.

New in version 1.0: the onerror callback

The ignore argument allows you to ignore certain modules, packages, or global objects during a scan. It should be a sequence containing strings and/or callables that will be used to match against the full dotted name of each object encountered during a scan. The sequence can contain any of these three types of objects:

- A string representing a full dotted name. To name an object by dotted name, use a string representing the full dotted name. For example, if you want to ignore the `my.package` package *and any of its subobjects or subpackages* during the scan, pass `ignore=['my.package']`.
- A string representing a relative dotted name. To name an object relative to the `package` passed to this method, use a string beginning with a dot. For example, if the `package` you've passed is imported as `my.package`, and you pass `ignore=['.mymodule']`, the `my.package.mymodule` mymodule *and any of its subobjects or subpackages* will be omitted during scan processing.
- A callable that accepts a full dotted name string of an object as its single positional argument and returns `True` or `False`. For example, if you want to skip all packages, modules, and global objects with a full dotted path that ends with the word "tests", you can use `ignore=[re.compile('tests$').search]`. If the callable returns `True` (or anything else truthy), the object is ignored, if it returns `False` (or anything else falsy) the object is not ignored. *Note that unlike string matches, ignores that use a callable don't cause submodules and subobjects of a module or package represented by a dotted name to also be ignored, they match individual objects found during a scan, including packages, modules, and global objects.*

You can mix and match the three types of strings in the list. For example, if the package being scanned is `my`, `ignore=['my.package', '.someothermodule', re.compile('tests$').search]` would cause `my.package` (and all its submodules and subobjects) to be ignored, `my.someothermodule` to be ignored, and any modules, packages, or global objects found during the scan that have a full dotted name that ends with the word `tests` to be ignored.

Note that packages and modules matched by any ignore in the list will not be imported, and their top-level code will not be run as a result.

A string or callable alone can also be passed as `ignore` without a surrounding list.

New in version 1.0a3: the `ignore` argument

`faust.utils.venusian.`**`attach`**(*fun: Callable, category: str, *, callback: Callable[[venusian.Scanner, str, Any], None] = None, **kwargs: Any*) → None

Shortcut for `venusian.attach()`.

This shortcut makes the callback argument optional.

Return type `None`

Terminal (TTY) Utilities

`faust.utils.terminal`

Terminal utilities.

`faust.utils.terminal.`**`isatty`**(*fh: IO*) → bool

Return True if fh has a controlling terminal.

Notes

Use with e.g. `sys.stdin`.

Return type `bool`

class faust.utils.terminal.**Spinner**(*file: IO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*) → None

Progress bar spinner.

bell = '\x08'

sprites = ['•', '◦', '●', 'O', '', '⊙', '']

cursor_hide = '\x1b[?25l'

cursor_show = '\x1b[?25h'

hide_cursor = True

stopped = False

update() → None

Draw spinner, single iteration.

Return type `None`

stop() → None

Stop spinner from being emitted.

Return type `None`

reset() → None

Reset state or allow restart.

Return type `None`

write(*s: str*) → None

Write spinner character to terminal.

Return type `None`

begin() → None

Prepare terminal for spinner starting.

Return type `None`

finish() → None

Finish spinner and reset terminal.

Return type `None`

class faust.utils.terminal.**SpinnerHandler**(*spinner: faust.utils.terminal.spinners.Spinner*, ***kwargs: Any*) → None

A logger handler that iterates our progress spinner for each log.

emit(*_record: logging.LogRecord*) → None

Emit the next spinner character.

Return type `None`

faust.utils.terminal.**Table**

alias of `terminaltables.base_table.BaseTable`

faust.utils.terminal.**TableDataT**

alias of `typing.Sequence`

faust.utils.terminal.**logtable**(*data: Sequence[Sequence[str]], *, title: str, target: IO = None, tty: bool = None, headers: Sequence[str] = None, **kwargs: Any*) → str

Prepare table for logging.

Will use ANSI escape codes if the log file is a tty.

Return type str

faust.utils.terminal.**table**(*data: Sequence[Sequence[str]], *, title: str, target: IO = None, tty: bool = None, **kwargs: Any*) → terminaltables.base_table.BaseTable

Create suitable terminaltables table for target.

Parameters

- **data** (*Sequence[Sequence[str]]*) – Table data.
- **target** (*IO*) – Target should be the destination output file for your table, and defaults to sys.stdout. ANSI codes will be used if the target has a controlling terminal, but not otherwise, which is why it's important to pass the correct output file.

Return type BaseTable

faust.utils.terminal.spinners

Terminal progress bar spinners.

class faust.utils.terminal.spinners.**Spinner**(*file: IO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*) → None

Progress bar spinner.

bell = '\x08'

sprites = ['•', '◦', '●', 'O', '', '⊙', '']

cursor_hide = '\x1b[?25l'

cursor_show = '\x1b[?25h'

hide_cursor = True

stopped = False

update() → None
Draw spinner, single iteration.

Return type None

stop() → None
Stop spinner from being emitted.

Return type None

reset() → None
Reset state or allow restart.

Return type None

write(*s: str*) → None
Write spinner character to terminal.

Return type None

begin() → None
Prepare terminal for spinner starting.

Return type None

finish() → None
Finish spinner and reset terminal.

Return type None

class faust.utils.terminal.spinners.**SpinnerHandler**(*spinner: faust.utils.terminal.spinners.Spinner*, ***kwargs: Any*) → None
A logger handler that iterates our progress spinner for each log.

emit(*_record: logging.LogRecord*) → None
Emit the next spinner character.

Return type None

faust.utils.terminal.tables

Using terminaltables to draw ANSI tables.

faust.utils.terminal.tables.**Table**
alias of terminaltables.base_table.BaseTable

faust.utils.terminal.tables.**TableDataT**
alias of typing.Sequence

faust.utils.terminal.tables.**table**(*data: Sequence[Sequence[str]], *, title: str, target: IO = None, tty: bool = None, **kwargs: Any*) → terminaltables.base_table.BaseTable
Create suitable terminaltables table for target.

Parameters

- **data** (*Sequence[Sequence[str]]*) – Table data.
- **target** (*IO*) – Target should be the destination output file for your table, and defaults to sys.stdout. ANSI codes will be used if the target has a controlling terminal, but not otherwise, which is why it's important to pass the correct output file.

Return type BaseTable

faust.utils.terminal.tables.**logtable**(*data: Sequence[Sequence[str]], *, title: str, target: IO = None, tty: bool = None, headers: Sequence[str] = None, **kwargs: Any*) → str
Prepare table for logging.

Will use ANSI escape codes if the log file is a tty.

Return type str

1.6.15 Web

faust.web.apps.graph

Web endpoint showing graph of running mode services.

class faust.web.apps.graph.**Graph**(*app: faust.types.app.AppT*, *web: faust.web.base.Web*) → None
Render image from graph of running services.

async get(*request: faust.web.base.Request*) → faust.web.base.Response
Draw image of the services running in this worker.

Return type *Response*

faust.web.apps.router

HTTP endpoint showing partition routing destinations.

class faust.web.apps.router.**TableList**(*app: faust.types.app.AppT*, *web: faust.web.base.Web*) → None
List routes for all tables.

async get(*request: faust.web.base.Request*) → faust.web.base.Response
Return JSON response with list of all table routes.

Return type *Response*

class faust.web.apps.router.**TableDetail**(*app: faust.types.app.AppT*, *web: faust.web.base.Web*) → None
List route for specific table.

async get(*request: faust.web.base.Request*, *name: str*) → faust.web.base.Response
Return JSON response with table metadata.

Return type *Response*

class faust.web.apps.router.**TableKeyDetail**(*app: faust.types.app.AppT*, *web: faust.web.base.Web*) → None
List information about key.

async get(*request: faust.web.base.Request*, *name: str*, *key: str*) → faust.web.base.Response
Return JSON response after looking up the route of a table key.

Parameters

- **name** (str) – Name of table.
- **key** (str) – Key to look up node for.

Raises **faust.web.exceptions.ServiceUnavailable** – if broker metadata has not yet been received.

Return type *Response*

faust.web.apps.stats

HTTP endpoint showing statistics from the Faust monitor.

class faust.web.apps.stats.**Stats**(*app: faust.types.app.AppT*, *web: faust.web.base.Web*) → None
Monitor statistics.

async get(*request: faust.web.base.Request*) → faust.web.base.Response
Return JSON response with sensor information.

Return type *Response*

class faust.web.apps.stats.**Assignment**(*app: faust.types.app.AppT*, *web: faust.web.base.Web*) → None
Cluster assignment information.

async get(*request: faust.web.base.Request*) → faust.web.base.Response
Return current assignment as a JSON response.

Return type *Response*

faust.web.base

Base interface for Web server and views.

class faust.web.base.**Response**
Web server response and status.

abstract property status
Return the response status code. :rtype: int

abstract property body
Return the response body as bytes. :rtype: bytes

abstract property headers
Return mapping of response HTTP headers. :rtype: MutableMapping[~KT, ~VT]

abstract property content_length
Return the size of the response body. :rtype: Optional[int]

abstract property content_type
Return the response content type. :rtype: str

abstract property charset
Return the response character set. :rtype: Optional[str]

abstract property chunked
Return True if response is chunked. :rtype: bool

abstract property compression
Return True if the response body is compressed. :rtype: bool

abstract property keep_alive
Return True if HTTP keep-alive enabled. :rtype: Optional[bool]

abstract property body_length
Size of HTTP response body. :rtype: int

class faust.web.base.**BlueprintManager**(*initial: Iterable[Tuple[str, Union[_T, str]]] = None*) → None
Manager of all blueprints.

add(*prefix: str, blueprint: Union[_T, str]*) → None
Register blueprint with this app.

Return type None

apply(*web: faust.web.base.Web*) → None
Apply all blueprints.

Return type None

class faust.web.base.**Web**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Web server and HTTP interface.

default_blueprints = [('/router', 'faust.web.apps.router:blueprint'), ('/table', 'faus

production_blueprints = [('', 'faust.web.apps.production_index:blueprint')]

debug_blueprints = [('/graph', 'faust.web.apps.graph:blueprint'), ('', 'faust.web.apps

content_separator = b'\r\n\r\n'

header_separator = b'\r\n'

header_key_value_separator = b': '

abstract text(*value: str*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create text response, using "text/plain" content-type.

Return type *Response*

abstract html(*value: str*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create HTML response from string, `text/html` content-type.

Return type *Response*

abstract json(*value: Any*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create new JSON response.

Accepts any JSON-serializable value and will automatically serialize it for you.

The content-type is set to "application/json".

Return type *Response*

abstract bytes(*value: bytes*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create new `bytes` response - for binary data.

Return type *Response*

abstract bytes_to_response(*s: bytes*) → faust.web.base.Response
Deserialize HTTP response from byte string.

Return type *Response*

abstract response_to_bytes(*response: faust.web.base.Response*) → bytes
Serialize HTTP response into byte string.

Return type *bytes*

abstract route(*pattern: str*, *handler: Callable*, *cors_options: Mapping[str, faust.types.web.ResourceOptions] = None*) → None
Add route for handler.

Return type None

abstract add_static(*prefix: str, path: Union[pathlib.Path, str], **kwargs: Any*) → None
Add static route.

Return type None

abstract async read_request_content(*request: faust.web.base.Request*) → bytes
Read HTTP body as bytes.

Return type *bytes*

abstract async wsgi() → Any
WSGI entry point.

Return type Any

add_view(*view_cls: Type[faust.types.web.View], *, prefix: str = '', cors_options: Mapping[str, faust.types.web.ResourceOptions] = None*) → faust.types.web.View
Add route for view.

Return type *View*

url_for(*view_name: str, **kwargs: Any*) → str
Get URL by view name.

If the provided view name has associated URL parameters, those need to be passed in as kwargs, or a TypeError will be raised.

Return type str

init_server() → None
Initialize and setup web server.

Return type None

property url
Return the canonical URL to this worker (including port). :rtype: URL

logger = <Logger faust.web.base (WARNING)>

class faust.web.base.**Request**(**args, **kwargs*)
HTTP Request.

abstract can_read_body() → bool
Return True if the request has a body.

Return type bool

abstract async read() → bytes
Read post data as bytes.

Return type bytes

abstract async text() → str
Read post data as text.

Return type str

abstract async json() → Any
Read post data and deserialize as JSON.

Return type Any

abstract async post() → Mapping[str, str]
Read post data.

Return type Mapping[str, str]

abstract property match_info
Return match info from URL route as a mapping. :rtype: `Mapping[str, str]`

abstract property query
Return HTTP query parameters as a mapping. :rtype: `Mapping[str, str]`

abstract property cookies
Return cookies as a mapping. :rtype: `Mapping[str, Any]`

faust.web.blueprints

Blueprints define reusable web apps.

They are lazy and need to be registered to an app to be activated:

```
from faust import web

blueprint = web.Blueprint('users')
cache = blueprint.cache(timeout=300.0)

@blueprint.route('/', name='list')
class UserListView(web.View):

    @cache.view()
    async def get(self, request: web.Request) -> web.Response:
        return web.json(...)

@blueprint.route('/{user_id}/', name='detail')
class UserDetailView(web.View):

    @cache.view(timeout=10.0)
    async def get(self,
                  request: web.Request,
                  user_id: str) -> web.Response:
        return web.json(...)
```

At this point the views are realized and can be used from Python code, but the cached `get` method handlers cannot be called yet.

To actually use the view from a web server, we need to register the blueprint to an app:

```
app = faust.App(
    'name',
    broker='kafka://',
    cache='redis://',
)

user_blueprint.register(app, url_prefix='/user/')
```

At this point the web server will have fully-realized views with actually cached method handlers.

The blueprint is registered with a prefix, so the URL for the `UserListView` is now `/user/`, and the URL for the `UserDetailView` is `/user/{user_id}/`.

Blueprints can be registered to multiple apps at the same time.

class `faust.web.blueprints.`**Blueprint**(*name: str*, **, url_prefix: Optional[str] = None*) → None
Define reusable web application.

view_name_separator = ':'

cache(*timeout: Union[datetime.timedelta, float, str] = None, include_headers: bool = False, key_prefix: str = None, backend: Union[Type[faust.types.web.CacheBackendT], str] = None*) → faust.types.web.CacheT
Cache API.

Return type *CacheT*

route(*uri: str, *, name: Optional[str] = None, cors_options: Mapping[str, faust.types.web.ResourceOptions] = None, base: Type[faust.types.web.View] = <class 'faust.types.web.View'>*) → Callable[Union[Type[faust.types.web.View], Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Union[Type[faust.types.web.View], Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]]
Create route by decorating handler or view class.

Return type Callable[[Union[Type[*View*], Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]], Union[Type[*View*], Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]]

static(*uri: str, file_or_directory: Union[str, pathlib.Path], *, name: Optional[str] = None*) → None
Add static route.

Return type None

register(*app: faust.types.app.AppT, *, url_prefix: Optional[str] = None*) → None
Register blueprint with app.

Return type None

init_webserver(*web: faust.types.web.Web*) → None
Init blueprint for web server start.

Return type None

on_webserver_init(*web: faust.types.web.Web*) → None
Call when web server starts.

Return type None

faust.web.cache

Caching.

class faust.web.cache.**Cache**(*timeout: Union[datetime.timedelta, float, str] = None, include_headers: bool = False, key_prefix: str = None, backend: Union[Type[faust.types.web.CacheBackendT], str] = None, **kwargs: Any*) → None
Cache interface.

ident = 'faustweb.cache.view'

view(*timeout: Union[datetime.timedelta, float, str] = None*, *include_headers: bool = False*, *key_prefix: str = None*, ***kwargs: Any*) → Callable[Callable, Callable]
Decorate view to be cached.

Return type Callable[[Callable], Callable]

async get_view(*key: str*, *view: faust.types.web.View*) → Optional[faust.types.web.Response]
Get cached value for HTTP view request.

Return type Optional[*Response*]

async set_view(*key: str*, *view: faust.types.web.View*, *response: faust.types.web.Response*, *timeout: Union[datetime.timedelta, float, str]*) → None
Set cached value for HTTP view request.

Return type None

can_cache_request(*request: faust.types.web.Request*) → bool
Return True if we can cache this type of HTTP request.

Return type bool

can_cache_response(*request: faust.types.web.Request*, *response: faust.types.web.Response*) → bool
Return True for HTTP status codes we CAN cache.

Return type bool

key_for_request(*request: faust.types.web.Request*, *prefix: str = None*, *method: str = None*, *include_headers: bool = False*) → str
Return a cache key created from web request.

Return type str

build_key(*request: faust.types.web.Request, method: str, prefix: str, headers: Mapping[str, str]*) → str
Build cache key from web request and environment.

Return type str

faust.web.cache.backends

Cache backend registry.

faust.web.cache.backends.**by_name**(*name: Union[_T, str]*) → _T

Return type ~_T

faust.web.cache.backends.**by_url**(*url: Union[str, yarl.URL]*) → _T
Get class associated with URL (scheme is used as alias key).

Return type ~_T

faust.web.cache.backends.base

Cache backend - base implementation.

class faust.web.cache.backends.base.**CacheBackend**(*app: faust.types.app.AppT*, *url: Union[yarl.URL, str] = 'memory://'*, ***kwargs: Any*) → None
Backend for cache operations.

logger = <Logger faust.web.cache.backends.base (WARNING)>

Unavailable
alias of *faust.web.cache.exceptions.CacheUnavailable*

operational_errors = ()

invalidating_errors = ()

irrecoverable_errors = ()

async get(*key: str*) → Optional[bytes]
Get cached-value by key.

Return type Optional[bytes]

async set(*key: str*, *value: bytes*, *timeout: float*) → None
Set cached-value by key.

Return type None

async delete(*key: str*) → None
Forget value for cache key.

Return type None

faust.web.cache.backends.memory

In-memory cache backend.

class faust.web.cache.backends.memory.**CacheStorage**
In-memory storage for cache.

get(*key: KT*) → Optional[VT]
Get value for key, or None if missing.

Return type Optional[~VT]

last_set_ttl(*key: KT*) → Optional[float]
Return the last set TTL for key, or None if missing.

Return type Optional[float]

expire(*key: KT*) → None
Expire value for key immediately.

Return type None

set(*key: KT*, *value: VT*) → None
Set value for key.

Return type None

setex(*key: KT*, *timeout: float*, *value: VT*) → None
Set value & set timeout for key.

Return type None

ttl(*key: KT*) → Optional[float]
Return the remaining TTL for key.

Return type Optional[float]

delete(*key: KT*) → None
Delete value for key.

Return type None

clear() → None
Clear all data.

Return type None

class faust.web.cache.backends.memory.**CacheBackend**(*app: faust.types.app.AppT*, *url: Union[yarl.URL, str] = 'memory://'*, ***kwargs: Any*) → None
In-memory backend for cache operations.

faust.web.cache.backends.redis

Redis cache backend.

class faust.web.cache.backends.redis.**RedisScheme**
Types of Redis configurations.

SINGLE_NODE = 'redis'

CLUSTER = 'rediscluster'

class faust.web.cache.backends.redis.**CacheBackend**(*app: faust.types.app.AppT*, *url: Union[yarl.URL, str]*, ***, *connect_timeout: float = None*, *stream_timeout: float = None*, *max_connections: int = None*, *max_connections_per_node: int = None*, ***kwargs: Any*) → None
Backend for cache operations using Redis.

async on_start() → None
Call when Redis backend starts.

Return type None

async connect() → None
Connect to Redis/Redis Cluster server.

Return type None

client
Return Redis client instance.

faust.web.cache.cache

Cache interface.

class faust.web.cache.cache.**Cache**(*timeout: Union[datetime.timedelta, float, str] = None*, *include_headers: bool = False*, *key_prefix: str = None*, *backend: Union[Type[faust.types.web.CacheBackendT], str] = None*, ***kwargs: Any*) → None
Cache interface.

ident = 'faustweb.cache.view'

view(*timeout: Union[datetime.timedelta, float, str] = None*, *include_headers: bool = False*, *key_prefix: str = None*, ***kwargs: Any*) → Callable[Callable, Callable]
Decorate view to be cached.

Return type Callable[[Callable], Callable]

async get_view(*key: str*, *view: faust.types.web.View*) → Optional[faust.types.web.Response]
Get cached value for HTTP view request.

Return type Optional[*Response*]

async set_view(*key: str*, *view: faust.types.web.View*, *response: faust.types.web.Response*, *timeout: Union[datetime.timedelta, float, str]*) → None
Set cached value for HTTP view request.

Return type None

can_cache_request(*request: faust.types.web.Request*) → bool
Return True if we can cache this type of HTTP request.

Return type bool

can_cache_response(*request: faust.types.web.Request*, *response: faust.types.web.Response*) → bool
Return True for HTTP status codes we CAN cache.

Return type bool

key_for_request(*request: faust.types.web.Request*, *prefix: str = None*, *method: str = None*, *include_headers: bool = False*) → str
Return a cache key created from web request.

Return type str

build_key(*request: faust.types.web.Request, method: str, prefix: str, headers: Mapping[str, str]*) → str
Build cache key from web request and environment.

Return type str

faust.web.cache.cache.**iri_to_uri**(*iri: str*) → str
Convert IRI to URI.

Return type str

faust.web.cache.exceptions

Cache-related errors.

exception faust.web.cache.exceptions.**CacheUnavailable**
The cache is currently unavailable.

faust.web.drivers

Web server driver registry.

faust.web.drivers.**by_name**(*name: Union[_T, str]*) → _T

Return type ~_T

faust.web.drivers.**by_url**(*url: Union[str, yarl.URL]*) → _T
Get class associated with URL (scheme is used as alias key).

Return type ~_T

faust.web.drivers.aiohttp

Web driver using aiohttp.

class faust.web.drivers.aiohttp.**Web**(*app: faust.types.app.AppT*, ***kwargs: Any*) → None
Web server and framework implementation using aiohttp.

driver_version = 'aiohttp=3.6.2'

handler_shutdown_timeout = 60.0

property cors
Return CORS config object. :rtype: CorsConfig

async on_start() → None
Call when the embedded web server starts.

Only used for *faust worker*, not when using *wsgi()*.

Return type None

async wsgi() → Any
Call WSGI handler.

Used by gunicorn and other WSGI compatible hosts to access the Faust web entry point.

Return type Any

text(*value: str*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create text response, using "text/plain" content-type.

Return type *Response*

html(*value: str*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create HTML response from string, text/html content-type.

Return type *Response*

json(*value: Any*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → Any
Create new JSON response.

Accepts any JSON-serializable value and will automatically serialize it for you.

The content-type is set to "application/json".

Return type Any

bytes(*value: bytes*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create new bytes response - for binary data.

Return type *Response*

async read_request_content(*request: faust.web.base.Request*) → bytes
Return the request body as bytes.

Return type *bytes*

route(*pattern: str*, *handler: Callable*, *cors_options: Mapping[str, aiohttp_cors.resource_options.ResourceOptions] = None*) → None
Add route for web view or handler.

Return type None

add_static(*prefix: str, path: Union[pathlib.Path, str], **kwargs: Any*) → None
Add route for static assets.

Return type None

bytes_to_response(*s: bytes*) → faust.web.base.Response
Deserialize byte string back into a response object.

Return type *Response*

response_to_bytes(*response: faust.web.base.Response*) → bytes
Convert response to serializable byte string.

The result is a byte string that can be deserialized using *bytes_to_response()*.

Return type *bytes*

async start_server() → None
Start the web server.

Return type None

async stop_server() → None
Stop the web server.

Return type None

logger = <Logger faust.web.drivers.aiohttp (WARNING)>

faust.web.exceptions

HTTP and related errors.

exception faust.web.exceptions.**WebError**(*detail: str = None, *, code: int = None, **extra_context: Any*) → None
Web related error.

Web related errors will have a status *code*, and a *detail* for the human readable error string.

It may also keep extra_context.

detail = 'Default not set on class'

code = None

exception faust.web.exceptions.**ServerError**(*detail: str = None, *, code: int = None, **extra_context: Any*) → None
Internal Server Error (500).

code = 500

detail = 'Internal server error.'

exception faust.web.exceptions.**ValidationError**(*detail: str = None, *, code: int = None, **extra_context: Any*) → None
Invalid input in POST data (400).

code = 400

detail = 'Invalid input.'

exception faust.web.exceptions.**ParseError**(*detail: str = None, *, code: int = None, **extra_context: Any*) → None
Malformed request (400).

code = 400

detail = 'Malformed request.'

exception faust.web.exceptions.**AuthenticationFailed**(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None

Incorrect authentication credentials (401).

code = 401

detail = 'Incorrect authentication credentials'

exception faust.web.exceptions.**NotAuthenticated**(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None

Authentication credentials were not provided (401).

code = 401

detail = 'Authentication credentials were not provided.'

exception faust.web.exceptions.**PermissionDenied**(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None

No permission to perform action (403).

code = 403

detail = 'You do not have permission to perform this action.'

exception faust.web.exceptions.**NotFound**(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None

Resource not found (404).

code = 404

detail = 'Not found.'

exception faust.web.exceptions.**MethodNotAllowed**(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None

HTTP Method not allowed (405).

code = 405

detail = 'Method not allowed.'

exception faust.web.exceptions.**NotAcceptable**(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None

Not able to satisfy the request `Accept` header (406).

code = 406

detail = 'Could not satisfy the request Accept header.'

exception faust.web.exceptions.**UnsupportedMediaType**(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None

Request contains unsupported media type (415).

code = 415

detail = 'Unsupported media type in request.'

exception faust.web.exceptions.**Throttled**(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None

Client is sending too many requests to server (429).

code = 429

detail = 'Request was throttled.'

faust.web.views

Class-based views.

class faust.web.views.**View**(*app: faust.types.app.AppT*, *web: faust.web.base.Web*) → None
Web view (HTTP endpoint).

exception ServerError(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None
Internal Server Error (500).

code = 500

detail = 'Internal server error.'

exception ValidationError(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None
Invalid input in POST data (400).

code = 400

detail = 'Invalid input.'

exception ParseError(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None
Malformed request (400).

code = 400

detail = 'Malformed request.'

exception NotAuthenticated(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None
Authentication credentials were not provided (401).

code = 401

detail = 'Authentication credentials were not provided.'

exception PermissionDenied(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None
No permission to perform action (403).

code = 403

detail = 'You do not have permission to perform this action.'

exception NotFound(*detail: str = None*, *, *code: int = None*, ***extra_context: Any*) → None
Resource not found (404).

code = 404

detail = 'Not found.'

classmethod from_handler(*fun: Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]*) → Type[faust.web.views.View]
Decorate `async def` handler function to create view.

Return type Type[*View*]

async dispatch(*request: Any*) → Any
Dispatch the request and perform any callbacks/cleanup.

Return type Any

async on_request_error(*request: faust.web.base.Request*, *exc: faust.web.exceptions.WebError*) → faust.web.base.Response
Call when a request raises an exception.

Return type *Response*

path_for(*view_name: str*, ***kwargs: Any*) → str
Return the URL path for view by name.

Supports match keyword arguments.

Return type str

url_for(*view_name: str*, *_base_url: Union[str, yarl.URL] = None*, ***kwargs: Any*) → yarl.URL
Return the canonical URL for view by name.

Supports match keyword arguments. Can take optional base name, which if not set will be the canonical URL of the app.

Return type URL

async head(*request: faust.web.base.Request*, ***kwargs: Any*) → Any
Override `head` to define the HTTP HEAD handler.

async get(*request: faust.web.base.Request*, ***kwargs: Any*) → Any
Override `get` to define the HTTP GET handler.

async post(*request: faust.web.base.Request*, ***kwargs: Any*) → Any
Override `post` to define the HTTP POST handler.

async put(*request: faust.web.base.Request*, ***kwargs: Any*) → Any
Override `put` to define the HTTP PUT handler.

async patch(*request: faust.web.base.Request*, ***kwargs: Any*) → Any
Override `patch` to define the HTTP PATCH handler.

async delete(*request: faust.web.base.Request*, ***kwargs: Any*) → Any
Override `delete` to define the HTTP DELETE handler.

async options(*request: faust.web.base.Request*, ***kwargs: Any*) → Any
Override `options` to define the HTTP OPTIONS handler.

async search(*request: faust.web.base.Request*, ***kwargs: Any*) → Any
Override `search` to define the HTTP SEARCH handler.

text(*value: str*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create text response, using "text/plain" content-type.

Return type *Response*

html(*value: str*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create HTML response from string, `text/html` content-type.

Return type *Response*

json(*value: Any*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create new JSON response.

Accepts any JSON-serializable value and will automatically serialize it for you.

The content-type is set to "application/json".

Return type *Response*

bytes(*value: bytes*, ***, *content_type: str = None*, *status: int = 200*, *reason: str = None*, *headers: MutableMapping = None*) → faust.web.base.Response
Create new `bytes` response - for binary data.

Return type *Response*

async read_request_content(*request: faust.web.base.Request*) → bytes
Return the request body as bytes.

Return type *bytes*

bytes_to_response(*s: bytes*) → faust.web.base.Response
Deserialize byte string back into a response object.

Return type *Response*

response_to_bytes(*response: faust.web.base.Response*) → bytes
Convert response to serializable byte string.

The result is a byte string that can be deserialized using *bytes_to_response()*.

Return type *bytes*

route(*pattern: str*, *handler: Callable*) → Any
Create new route from pattern and handler.

Return type Any

notfound(*reason: str = 'Not Found'*, ***kwargs: Any*) → faust.web.base.Response
Create not found error response.

Deprecated: Use `raise self.NotFound()` instead.

Return type *Response*

error(*status: int*, *reason: str*, ***kwargs: Any*) → faust.web.base.Response
Create error JSON response.

Return type *Response*

faust.web.views.**takes_model**(*Model: Type[faust.types.models.ModelT]*) → Callable[Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]]
Decorate view function to return model data.

Return type Callable[[Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]], Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]]

faust.web.views.**gives_model**(*Model: Type[faust.types.models.ModelT]*) → Callable[Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]], Union[Callable[[faust.types.web.View, faust.types.web.Request], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]], Callable[[faust.types.web.View, faust.types.web.Request, Any, Any], Union[Coroutine[[Any, Any], faust.types.web.Response], Awaitable[faust.types.web.Response]]]]]

Decorate view function to automatically decode POST data.

The POST data is decoded using the model you specify.

Return type Callable[[Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]], Union[Callable[[*View*, *Request*], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]], Callable[[*View*, *Request*, Any, Any], Union[Coroutine[Any, Any, *Response*], Awaitable[*Response*]]]]]

1.6.16 CLI

faust.cli.agents

Program `faust agents` used to list agents.

class faust.cli.agents.**agents**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None

List agents.

title = 'Agents'

headers = ['name', 'topic', 'help']

sortkey = operator.attrgetter('name')

options = [option('--local/--no-local', help='Include agents using a local channel')]

async run(*local: bool*) → None

Dump list of available agents in this application.

Return type None

agents(*, *local: bool = False*) → Sequence[faust.types.agents.AgentT]

Convert list of agents to terminal table rows.

Return type Sequence[*AgentT*[]]

agent_to_row(*agent: faust.types.agents.AgentT*) → Sequence[str]

Convert agent fields to terminal table row.

Return type Sequence[str]

faust.cli.base

Command-line programs using click.

class faust.cli.base.**argument**(**args: Any*, ***kwargs: Any*)
Create command-line argument.

SeeAlso: click.argument()

class faust.cli.base.**option**(**args: Any*, *show_default: bool = True*, ***kwargs: Any*)
Create command-line option.

SeeAlso: click.option()

faust.cli.base.**find_app**(*app: str*, ***, *symbol_by_name: Callable = <function symbol_by_name>*, *imp: Callable = <function import_from_cwd>*, *attr_name: str = 'app'*) → faust.types.app.AppT
Find app by string like examples.simple.

Notes

This function uses import_from_cwd to temporarily add the current working directory to PYTHONPATH, such that when importing the app it will search the current working directory last.

You can think of it as temporarily running with the PYTHONPATH set like this:

You can disable this with the imp keyword argument, for example passing imp=importlib.import_module.

Examples

```
>>> # If providing the name of a module, it will attempt
>>> # to find an attribute name [app] in that module.
>>> # Example below is the same as importing:
>>> #     from examples.simple import app
>>> find_app('examples.simple')
```

```
>>> # If you want an attribute other than "app" you can
>>> # use : to separate module and attribute.
>>> # Example below is the same as importing:
>>> #     from examples.simple import my_app
>>> find_app('examples.simple:my_app')
```

```
>>> # You can also use . for the module/attribute separator.
>>> find_app('examples.simple.my_app')
```

Return type *AppT*[]

class faust.cli.base.**Command**(*ctx: click.core.Context*, **args: Any*, ***kwargs: Any*) → None
Base class for subcommands.

exception UsageError(*message*, *ctx=None*)
An internal exception that signals a usage error. This typically aborts any further handling.

Parameters

- **message** – the error message to display.

- **ctx** – optionally the context that caused this error. Click will fill in the context automatically in some situations.

exit_code = 2

show(*file=None*)

abstract = True

daemon = False

redirect_stdouts = None

redirect_stdouts_level = None

builtin_options = [<function version_option.<locals>.decorator>, option('--app', '-A',

options = None

classmethod as_click_command() → Callable
Convert command into click command.

Return type Callable

classmethod parse(*argv: Sequence[str]*) → Mapping
Parse command-line arguments in `argv` and return mapping.

Return type Mapping[~KT, +VT_co]

prog_name = ''

async run(**args: Any*, ***kwargs: Any*) → Any
Override this method to define what your command does.

async execute(**args: Any*, ***kwargs: Any*) → Any
Execute command.

Return type Any

async on_stop() → None
Call after command executed.

Return type None

run_using_worker(**args: Any*, ***kwargs: Any*) → NoReturn
Execute command using *faust.Worker*.

Return type _NoReturn

on_worker_created(*worker: mode.worker.Worker*) → None
Call when creating *faust.Worker* to execute this command.

Return type None

as_service(*loop: asyncio.events.AbstractEventLoop*, **args: Any*, ***kwargs: Any*) → mode.services.Service
Wrap command in a mode.Service object.

Return type Service[]

worker_for_service(*service: mode.types.services.ServiceT*, *loop: asyncio.events.AbstractEventLoop = None*) → mode.worker.Worker
Create *faust.Worker* instance for this command.

Return type Worker[]

tabulate(*data: Sequence[Sequence[str]], headers: Sequence[str] = None, wrap_last_row: bool = True, title: str = '', title_color: str = 'blue', **kwargs: Any*) → str
Create an ANSI representation of a table of two-row tuples.

See also:

Keyword arguments are forwarded to `terminaltables.SingleTable`

Note: If the `--json` option is enabled this returns json instead.

Return type `str`

table(*data: Sequence[Sequence[str]], title: str = '', **kwargs: Any*) → terminaltables.base_table.BaseTable
Format table data as ANSI/ASCII table.

Return type `BaseTable`

color(*name: str, text: str*) → str
Return text having a certain color by name.

Examples::

```
>>> self.color('blue', 'text_to_color')
>>> self.color('hiblue', text_to_color')
```

See also:

colorclass: for a list of available colors.

Return type `str`

dark(*text: str*) → str
Return cursor text.

Return type `str`

bold(*text: str*) → str
Return text in bold.

Return type `str`

bold_tail(*text: str, *, sep: str = '.'*) → str
Put bold emphasis on the last part of a `foo.bar.baz` string.

Return type `str`

say(*message: str, file: IO = None, err: IO = None, **kwargs: Any*) → None
Print something to stdout (or use `file=stderr` kwarg).

Note: Does not do anything if the `--quiet` option is enabled.

Return type `None`

carp(*s: Any, **kwargs: Any*) → None
Print something to stdout (or use `file=stderr` kwargs).

Note: Does not do anything if the `--debug` option is enabled.

Return type None

dumps(*obj: Any*) → str
Serialize object using JSON.

Return type str

property loglevel
Return the log level used for this command. :rtype: str

property blocking_timeout
Return the blocking timeout used for this command. :rtype: float

property console_port
Return the aiomonitor console port. :rtype: int

class faust.cli.base.**AppCommand**(*ctx: click.core.Context, *args: Any, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, **kwargs: Any*) → None
Command that takes `-A app` as argument.

abstract = False

require_app = True

value_serialier = None
The *codec* used to serialize values. Taken from instance parameters or `value_serializer`.

classmethod from_handler(**options: Any, **kwargs: Any*) → Callable[Callable, Type[faust.cli.base.AppCommand]]
Decorate `async def` command to create command class.

Return type Callable[[Callable], Type[*AppCommand*]]

key_serializer = None
The *codec* used to serialize keys. Taken from instance parameters or `key_serializer`.

async on_stop() → None
Call after command executed.

Return type None

to_key(*typ: Optional[str], key: str*) → Any
Convert command-line argument string to model (key).

Parameters

- **typ** (Optional[str]) – The name of the model to create.
- **key** (str) – The string json of the data to populate it with.

Notes

Uses *key_serializer* to set the *codec* for the key (e.g. "json"), as set by the *--key-serializer* option.

Return type Any

to_value(*typ: Optional[str], value: str*) → Any
Convert command-line argument string to model (value).

Parameters

- **typ** (Optional[str]) – The name of the model to create.
- **key** – The string json of the data to populate it with.

Notes

Uses value_serializer to set the *codec* for the value (e.g. "json"), as set by the *--value-serializer* option.

Return type Any

to_model(*typ: Optional[str], value: str, serializer: Union[faust.types.codecs.CodecT, str, None]*) → Any
Convert command-line argument to model.

Generic version of *to_key()*/*to_value()*.

Parameters

- **typ** (Optional[str]) – The name of the model to create.
- **key** – The string json of the data to populate it with.
- **serializer** (Union[*CodecT*, str, None]) – The argument setting it apart from to_key/to_value enables you to specify a custom serializer not mandated by *key_serializer*, and value_serializer.

Notes

Uses value_serializer to set the *codec* for the value (e.g. "json"), as set by the *--value-serializer* option.

Return type Any

import_relative_to_app(*attr: str*) → Any
Import string like "module.Model", or "Model" to model class.

Return type Any

to_topic(*entity: str*) → Any
Convert topic name given on command-line to app.topic().

Return type Any

abbreviate_fqdn(*name: str*, *, *prefix: str = ''*) → str
Abbreviate fully-qualified Python name, by removing origin.

app.conf.origin is the package where the app is defined, so if this is examples.simple it returns the truncated:

```
>>> app.conf.origin
'examples.simple'
>>> abbr_fqdn(app.conf.origin,
...           'examples.simple.Withdrawal',
...           prefix='[...]')
'[...]Withdrawal'
```

but if the package is not part of origin it provides the full path:

```
>>> abbr_fqdn(app.conf.origin,
...           'examples.other.Foo', prefix='[...]')
'examples.other.foo'
```

Return type str

faust.cli.clean_versions

Program `faust reset` used to delete local table state.

class faust.cli.clean_versions.**clean_versions**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None

Delete old version directories.

> **Warning:** This command will result in the destruction of the following files:
>
> 1) Table data for previous versions of the app.

async run() → None

Execute command.

Return type None

remove_old_versiondirs() → None

Remove data from old application versions from data directory.

Return type None

faust.cli.completion

completion - Command line utility for completion.

Supports `bash`, `ksh`, `zsh`, etc.

class faust.cli.completion.**completion**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None

Output shell completion to be evaluated by the shell.

require_app = False

async run() → None
Dump click completion script for Faust CLI.

Return type None

shell() → str
Return the current shell used in this environment.

Return type str

faust.cli.faust

Program `faust` (umbrella command).

class faust.cli.faust.**agents**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
List agents.

```
title = 'Agents'
headers = ['name', 'topic', 'help']
sortkey = operator.attrgetter('name')
options = [option('--local/--no-local', help='Include agents using a local channel')]
```

async run(*local: bool*) → None
Dump list of available agents in this application.

Return type None

agents(*, *local: bool = False*) → Sequence[faust.types.agents.AgentT]
Convert list of agents to terminal table rows.

Return type Sequence[*AgentT*[]]

agent_to_row(*agent: faust.types.agents.AgentT*) → Sequence[str]
Convert agent fields to terminal table row.

Return type Sequence[str]

faust.cli.faust.**call_command**(*command: str*, *args: List[str] = None*, *stdout: IO = None*, *stderr: IO = None*, *side_effects: bool = False*, ***kwargs: Any*) → Tuple[int, IO, IO]

Return type Tuple[int, IO[AnyStr], IO[AnyStr]]

class faust.cli.faust.**clean_versions**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
Delete old version directories.

> **Warning:** This command will result in the destruction of the following files:
>
> 1) Table data for previous versions of the app.

async run() → None
Execute command.

Return type None

remove_old_versiondirs() → None
Remove data from old application versions from data directory.

Return type None

class faust.cli.faust.**completion**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
Output shell completion to be evaluated by the shell.

require_app = False

async run() → None
Dump click completion script for Faust CLI.

Return type None

shell() → str
Return the current shell used in this environment.

Return type str

class faust.cli.faust.**livecheck**(**args: Any*, ***kwargs: Any*) → None
Manage LiveCheck instances.

class faust.cli.faust.**model**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
Show model detail.

headers = ['field', 'type', 'default']

options = [argument('name')]

async run(*name: str*) → None
Dump list of registered models to terminal.

Return type None

model_fields(*model: Type[faust.types.models.ModelT]*) → Sequence[Sequence[str]]
Convert model fields to terminal table rows.

Return type Sequence[Sequence[str]]

field(*field: faust.types.models.FieldDescriptorT*) → Sequence[str]
Convert model field model to terminal table columns.

Return type Sequence[str]

model_to_row(*model: Type[faust.types.models.ModelT]*) → Sequence[str]
Convert model to terminal table row.

Return type Sequence[str]

class faust.cli.faust.**models**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
List all available models as a tabulated list.

title = 'Models'

headers = ['name', 'help']

sortkey = operator.attrgetter('_options.namespace')

options = [option('--builtins/--no-builtins', default=False)]

async run(*, *builtins: bool*) → None
Dump list of available models in this application.

Return type None

models(*builtins: bool*) → Sequence[Type[faust.types.models.ModelT]]
Convert list of models to terminal table rows.

Return type Sequence[Type[*ModelT*]]

model_to_row(*model: Type[faust.types.models.ModelT]*) → Sequence[str]
Convert model fields to terminal table columns.

Return type Sequence[str]

class faust.cli.faust.**reset**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
Delete local table state.

Warning: This command will result in the destruction of the following files:

1) The local database directories/files backing tables (does not apply if an in-memory store like memory:// is used).

Notes

This data is technically recoverable from the Kafka cluster (if intact), but it'll take a long time to get the data back as you need to consume each changelog topic in total.

It'd be faster to copy the data from any standbys that happen to have the topic partitions you require.

async run() → None
Execute command.

Return type None

async reset_tables() → None
Reset local state for all tables.

Return type None

class faust.cli.faust.**send**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
Send message to agent/topic.

options = [option('--key-type', '-K', help='Name of model to serialize key into.'), op

async run(*entity: str*, *value: str*, **args: Any*, *key: str = None*, *key_type: str = None*, *key_serializer: str = None*, *value_type: str = None*, *value_serializer: str = None*, *partition: int = 1*, *timestamp: float = None*, *repeat: int = 1*, *min_latency: float = 0.0*, *max_latency: float = 0.0*, ***kwargs: Any*) → Any
Send message to topic/agent/channel.

Return type Any

class faust.cli.faust.**tables**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None

List available tables.

title = 'Tables'

async run() → None

Dump list of application tables to terminal.

Return type None

class faust.cli.faust.**worker**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None

Start worker instance for given app.

daemon = True

redirect_stdouts = True

worker_options = [option('--with-web/--without-web', default=True, help='Enable/disabl

options = [option('--with-web/--without-web', default=True, help='Enable/disable web s

on_worker_created(*worker: mode.worker.Worker*) → None

Print banner when worker starts.

Return type None

as_service(*loop: asyncio.events.AbstractEventLoop*, **args: Any*, ***kwargs: Any*) → mode.types.services.ServiceT

Return the service this command should execute.

For the worker we simply start the application itself.

Note: The application will be started using a *faust.Worker*.

Return type ServiceT[]

banner(*worker: mode.worker.Worker*) → str

Generate the text banner emitted before the worker starts.

Return type str

faust_ident() → str

Return Faust version information as ANSI string.

Return type str

platform() → str

Return platform identifier as ANSI string.

Return type str

faust.cli.livecheck

Program `faust worker` used to start application from console.

class faust.cli.livecheck.**livecheck**(**args: Any*, ***kwargs: Any*) → None
Manage LiveCheck instances.

faust.cli.model

Program `faust model` used to list details about a model.

class faust.cli.model.**model**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
Show model detail.

headers = ['field', 'type', 'default']

options = [argument('name')]

async run(*name: str*) → None
Dump list of registered models to terminal.

Return type None

model_fields(*model: Type[faust.types.models.ModelT]*) → Sequence[Sequence[str]]
Convert model fields to terminal table rows.

Return type Sequence[Sequence[str]]

field(*field: faust.types.models.FieldDescriptorT*) → Sequence[str]
Convert model field model to terminal table columns.

Return type Sequence[str]

model_to_row(*model: Type[faust.types.models.ModelT]*) → Sequence[str]
Convert model to terminal table row.

Return type Sequence[str]

faust.cli.models

Program `faust models` used to list models available.

class faust.cli.models.**models**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
List all available models as a tabulated list.

title = 'Models'

headers = ['name', 'help']

sortkey = operator.attrgetter('_options.namespace')

options = [option('--builtins/--no-builtins', default=False)]

async run(***, *builtins: bool*) → None
Dump list of available models in this application.

Return type None

models(*builtins: bool*) → Sequence[Type[faust.types.models.ModelT]]
Convert list of models to terminal table rows.

Return type Sequence[Type[*ModelT*]]

model_to_row(*model: Type[faust.types.models.ModelT]*) → Sequence[str]
Convert model fields to terminal table columns.

Return type Sequence[str]

faust.cli.params

Python click parameter types.

class faust.cli.params.**CaseInsensitiveChoice**(*choices: Iterable[Any]*)
Case-insensitive version of click.Choice.

convert(*value: str, param: Optional[click.core.Parameter], ctx: Optional[click.core.Context]*) → Any
Convert string to case-insensitive choice.

Return type Any

class faust.cli.params.**TCPPort**
CLI option: TCP Port (integer in range 1 - 65535).

name = 'range[1-65535]'

class faust.cli.params.**URLParam** → None
URL click parameter type.

Converts any string URL to yarl.URL.

name = 'URL'

convert(*value: str, param: Optional[click.core.Parameter], ctx: Optional[click.core.Context]*) → yarl.URL
Convert str argument to yarl.URL.

Return type URL

faust.cli.reset

Program faust reset used to delete local table state.

class faust.cli.reset.**reset**(*ctx: click.core.Context, *args: Any, key_serializer: Union[faust.types.codecs.CodecT, str, None] = None, value_serializer: Union[faust.types.codecs.CodecT, str, None] = None, **kwargs: Any*) → None
Delete local table state.

> **Warning:** This command will result in the destruction of the following files:
>
> 1) The local database directories/files backing tables (does not apply if an in-memory store like memory:// is used).

Notes

This data is technically recoverable from the Kafka cluster (if intact), but it'll take a long time to get the data back as you need to consume each changelog topic in total.

It'd be faster to copy the data from any standbys that happen to have the topic partitions you require.

async run() → None
Execute command.

Return type None

async reset_tables() → None
Reset local state for all tables.

Return type None

faust.cli.send

Program `faust send` used to send events to agents and topics.

class faust.cli.send.**send**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
Send message to agent/topic.

options = [option('--key-type', '-K', help='Name of model to serialize key into.'), op

async run(*entity: str*, *value: str*, **args: Any*, *key: str = None*, *key_type: str = None*, *key_serializer: str = None*, *value_type: str = None*, *value_serializer: str = None*, *partition: int = 1*, *timestamp: float = None*, *repeat: int = 1*, *min_latency: float = 0.0*, *max_latency: float = 0.0*, ***kwargs: Any*) → Any
Send message to topic/agent/channel.

Return type Any

faust.cli.tables

Program `faust tables` used to list tables.

class faust.cli.tables.**tables**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None
List available tables.

title = 'Tables'

async run() → None
Dump list of application tables to terminal.

Return type None

faust.cli.worker

Program `faust worker` used to start application from console.

class faust.cli.worker.**worker**(*ctx: click.core.Context*, **args: Any*, *key_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, *value_serializer: Union[faust.types.codecs.CodecT, str, None] = None*, ***kwargs: Any*) → None

Start worker instance for given app.

daemon = True

redirect_stdouts = True

worker_options = [option('--with-web/--without-web', default=True, help='Enable/disabl

options = [option('--with-web/--without-web', default=True, help='Enable/disable web s

on_worker_created(*worker: mode.worker.Worker*) → None

Print banner when worker starts.

Return type None

as_service(*loop: asyncio.events.AbstractEventLoop*, **args: Any*, ***kwargs: Any*) → mode.types.services.ServiceT

Return the service this command should execute.

For the worker we simply start the application itself.

Note: The application will be started using a *faust.Worker*.

Return type ServiceT[]

banner(*worker: mode.worker.Worker*) → str

Generate the text banner emitted before the worker starts.

Return type str

faust_ident() → str

Return Faust version information as ANSI string.

Return type str

platform() → str

Return platform identifier as ANSI string.

Return type str

1.7 Changes

This document contain change notes for bugfix releases in the Faust 1.9 series. If you're looking for previous releases, please visit the *History* section.

1.7.1 1.9.0

release-date 2019-10-29 2:35 A.M PST

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on robinhood-aiokafka 1.1.3

    Several issues fixed by Vikram Patki (@patkivikram):

    * https://github.com/robinhood/aiokafka/pull/14
    * https://github.com/robinhood/aiokafka/pull/12
    * https://github.com/robinhood/aiokafka/pull/11
    * https://github.com/robinhood/aiokafka/pull/10
  - Now depends on Mode 4.1.3.
- **Stream**: `group_by` no longer repeats prefix in autogenerated repartition topic name.

  This change is backwards incompatible as the name of repartition topics will change.
- **Consumer**: Fixed livelock offset not advancing message (Issue #450).

  Contributed by Dhruva Patil (@DhruvaPatil98).
- **Consumer**: Fixed Out of Memory error when gaps in topic offsets are very large.

  Fix contributed by Vikram Patki (@patkivikram).
- **Table**: Windowed tables now supports an `on_window_close` callback that is called whenever a window expires (Issue #446).

  For an example see `examples/windowed_aggregation.py` in the Faust source distribution.

  Contributed by Leandro Vonwerra (@lvwerra).
- **Table**: Fixed issue with worker startup crash when global tables are used.
- **Web**: Views now support the `SEARCH` HTTP method (Issue #460).

  Contributed by Ignacio Peluffo (@ipeluffo).
- **Table**: Global table changes are now instantly reflected on other nodes (Issue #451).

  The default `standby_buffer_size` setting for tables was set to 1000 also for global tables, which means changes will only be seen in batches of thousands. We now set this to 1 for global tables, so changes are applied immediately as they arrive.
- **Opentracing**: Trace category now includes application name.

  This means that where before the categories were `_aiokafka` and `_faust` these are now `{app_name}-_aiokafka` and `{app_name}-_faust`.
- **Opentracing**: Traces for aiokafka Kafka rebalances are now consolidated so traces for the same generation are grouped together.

  This change also adds a lot more context for spans in the rebalancing trace.
- **Documentation** fixes by:
  - Archit Dwivedi (@archit0).

1.8 Contributing

Welcome!

This document is fairly extensive and you aren't really expected to study this in detail for small contributions;

> The most important rule is that contributing must be easy and that the community is friendly and not nit-picking on details, such as coding style.

If you're reporting a bug you should read the Reporting bugs section below to ensure that your bug report contains enough information to successfully diagnose the issue, and if you're contributing code you should try to mimic the conventions you see surrounding the code you're working on, but in the end all patches will be cleaned up by the person merging the changes so don't worry too much.

1.8.1 Code of Conduct

Everyone interacting in the project's code bases, issue trackers, chat rooms, and mailing lists is expected to follow the Faust Code of Conduct.

As contributors and maintainers of these projects, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in these projects a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, or nationality.

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks
- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission
- Other unethical or unprofessional conduct.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct. By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This code of conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by opening an issue or contacting one or more of the project maintainers.

This Code of Conduct is adapted from the Contributor Covenant, version 1.2.0 available at http://contributor-covenant.org/version/1/2/0/.

1.8.2 Reporting Bugs

Security

You must never report security related issues, vulnerabilities or bugs including sensitive information to the bug tracker, or elsewhere in public. Instead sensitive bugs must be sent by email to `security@celeryproject.org`.

If you'd like to submit the information encrypted our PGP key is:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.15 (Darwin)

mQENBFJpWDkBCADFIc9/Fpgse4owLNvsTC7GYfnJL19XO0hnL99sPx+DPbfr+cSE
9wiU+Wp2TfUX7pCLEGrODiEP6ZCZbgtiPgId+JYvMxpP6GXbjiIlHRw1EQNH8RlX
cVxy3rQfVv8PGGiJuyBBjxzvETHW25htVAZ5TI1+CkxmuyyEYqgZN2fNd0wEU19D
+c10G1gSECbCQTCbacLSzdpngAt1Gkrc96r7wGHBBSvDaGDD2pFSkVuTLMbIRrVp
lnKOPMsUijiip2EMr2DvfuXiUIUvaqInTPNWkDynLoh69ib5xC19CSVLONjkKBsr
Pe+qAY29liBatatpXsydY7GIUzyBT3MzgMJlABEBAAG0MUNlbGVyeSBTZWN1cml0
eSBUZWFtIDxzZWN1cml0eUBjZWxlcnlwcm9qZWN0Lm9yZz6JATgEEwECACIFAlJp
WDkCGwMGCwkIBwMCBhUIAgkKCwQWAgMBAh4BAheAAAoJEOArFOUDCicIw1IH/26f
CViDC7/P13jr+srRdjAsWvQztia9HmTlY8cUnbmkR9w6b6j3F2ayw8VhkyFWgYEJ
wtPBv8mHKADiVSFARS+0yGsfCkia5wDSQuIv6XqRlIrXUyqJbmF4NUFTyCZYoh+C
ZiQpN9xGhFPr5QDlMx2izWg1rvWlG1jY2Es1v/xED3AeCOB1eUGvRe/uJHKjGv7J
rj0pFcptZX+WDF22AN235WYwgJM6TrNfSu8sv8vNAQOVnsKcgsqhuwomSGsOfMQj
LFzIn95MKBBU1G5wOs7JtwiV9jefGqJGBO2FAvOVbvPdK/saSnB+7K36dQcIHqms
5hU4Xj0RIJiod5idlRC5AQ0EUmlYOQEIAJs8OwHMkrdcvy9kk2HBVbdqhgAREMKy
gmphDp7prRL9FqSY/dKpCbG0u82zyJypdb7QiaQ5pfPzPpQcd2dIcohkkh7G3E+e
hS2L9AXHpwR26/PzMBXyr2iNnNc4vTksHvGVDxzFnRpka6vbI/hrrZmYNYh9EAiv
uhE54b3/XhXwFgHjZXb9i8hgJ3nsO0pRwvUAM1bRGMbvf8e9F+kqgV0yWYNnh6QL
4Vpl1+epqp2RKPHyNQftbQyrAHXT9kQF9pPlx013MKYaFTADscuAp4T3dy7xmiwS
crqMbZLzfrxfFOsNxTUGE5vmJCcm+mybAtRo4aV6ACohAO9NevMx8pUAEQEAAYkB
HwQYAQIACQUCUmlYOQIbDAAKCRDgKxTlAwonCNFbB/9esir/f7TufE+isNqErzR/
aZKZo2WzZR9c75kbqo6J6DYuUHe6xI0OZ2qZ60iABDEZAiNXGulysFLCiPdatQ8x
8zt3DF9BMkEck54ZvAjpNSern6zfZb1jPYWZq3TKxlTs/GuCgBAuV4i5vDTZ7xK/
aF+OFY5zN7ciZHkqLgMiTZ+RhqRcK6FhVBP/Y7d9NlBOcDBTxxE1ZO1ute6n7guJ
ciw4hfoRk8qNN19szZuq3UU64zpkM2sBsIFM9tGF2FADRxiOaOWZHmIyVZriPFqW
RUwjSjs7jBVNq0Vy4fCu/5+e+XLOUBOoqtM5W7ELt0t1w9tXebtPEetV86in8fU2
=0chn
-----END PGP PUBLIC KEY BLOCK-----
```

Other bugs

Bugs can always be described to the *Mailing list*, but the best way to report an issue and to ensure a timely response is to use the issue tracker.

1) **Create a GitHub account**.

You need to create a GitHub account to be able to create new issues and participate in the discussion.

2) **Determine if your bug is really a bug**.

You shouldn't file a bug if you're requesting support. For that you can use the *Mailing list*, or *Slack*.

3) **Make sure your bug hasn't already been reported**.

Search through the appropriate Issue tracker. If a bug like yours was found, check if you have new information that could be reported to help the developers fix the bug.

4) **Check if you're using the latest version**.

A bug could be fixed by some other improvements and fixes - it might not have an existing report in the bug tracker. Make sure you're using the latest release of Faust.

5) **Collect information about the bug**.

To have the best chance of having a bug fixed, we need to be able to easily reproduce the conditions that caused it. Most of the time this information will be from a Python traceback message, though some bugs might be in design, spelling or other errors on the website/docs/code.

A) If the error is from a Python traceback, include it in the bug report.

B) We also need to know what platform you're running (Windows, macOS, Linux, etc.), the version of your Python interpreter, and the version of Faust, and related packages that you were running when the bug occurred.

C) If you're reporting a race condition or a deadlock, tracebacks can be hard to get or might not be that useful. Try to inspect the process to get more diagnostic data. Some ideas:

- Collect tracing data using *strace`_(Linux), :command:`dtruss* (macOS), and **`ktrace`** (BSD), ltrace, and lsof.

D) Include the output from the **`faust report`** command:

```
$ faust -A proj report
```

This will also include your configuration settings and it try to remove values for keys known to be sensitive, but make sure you also verify the information before submitting so that it doesn't contain confidential information like API tokens and authentication credentials.

6) **Submit the bug**.

By default GitHub will email you to let you know when new comments have been made on your bug. In the event you've turned this feature off, you should check back on occasion to ensure you don't miss any questions a developer trying to fix the bug might ask.

Issue Trackers

Bugs for a package in the Faust ecosystem should be reported to the relevant issue tracker.

- Faust - https://github.com/robinhood/faust/issues
- Mode - https://github.com/ask/mode/issues

If you're unsure of the origin of the bug you can ask the *Mailing list*, or just use the Faust issue tracker.

1.8.3 Contributors guide to the code base

There's a separate section for internal details, including details about the code base and a style guide.

Read *Developer Guide* for more!

1.8.4 Versions

Version numbers consists of a major version, minor version and a release number. Faust uses the versioning semantics described by SemVer: http://semver.org.

Stable releases are published at PyPI while development releases are only available in the GitHub git repository as tags. All version tags starts with "v", so version 0.8.0 is the tag v0.8.0.

1.8.5 Branches

Current active version branches:

- dev (which git calls "master") (https://github.com/robinhood/faust/tree/master)
- 1.0 (https://github.com/robinhood/faust/tree/1.0)

You can see the state of any branch by looking at the Changelog:

https://github.com/robinhood/faust/blob/master/Changelog.rst

If the branch is in active development the topmost version info should contain meta-data like:

```
2.4.0
======
:release-date: TBA
:status: DEVELOPMENT
:branch: dev (git calls this master)
```

The `status` field can be one of:

- `PLANNING`

  The branch is currently experimental and in the planning stage.

- `DEVELOPMENT`

  The branch is in active development, but the test suite should be passing and the product should be working and possible for users to test.

- `FROZEN`

  The branch is frozen, and no more features will be accepted. When a branch is frozen the focus is on testing the version as much as possible before it is released.

dev branch

The dev branch (called "master" by git), is where development of the next version happens.

Maintenance branches

Maintenance branches are named after the version – for example, the maintenance branch for the 2.2.x series is named `2.2`.

Previously these were named `releaseXX-maint`.

The versions we currently maintain is:

- 1.0

  This is the current series.

Archived branches

Archived branches are kept for preserving history only, and theoretically someone could provide patches for these if they depend on a series that's no longer officially supported.

An archived version is named `X.Y-archived`.

Our currently archived branches are:

We don't currently have any archived branches.

Feature branches

Major new features are worked on in dedicated branches. There's no strict naming requirement for these branches.

Feature branches are removed once they've been merged into a release branch.

1.8.6 Tags

- Tags are used exclusively for tagging releases. A release tag is named with the format `vX.Y.Z` – for example `v2.3.1`.
- Experimental releases contain an additional identifier `vX.Y.Z-id` – for example `v3.0.0-rc1`.
- Experimental tags may be removed after the official release.

1.8.7 Working on Features & Patches

Note: Contributing to Faust should be as simple as possible, so none of these steps should be considered mandatory.

You can even send in patches by email if that's your preferred work method. We won't like you any less, any contribution you make is always appreciated!

However following these steps may make maintainers life easier, and may mean that your changes will be accepted sooner.

Forking and setting up the repository

Create your fork

First you need to fork the Faust repository, a good introduction to this is in the GitHub Guide: Fork a Repo.

After you have cloned the repository you should checkout your copy to a directory on your machine:

```
$ git clone git@github.com:username/faust.git
```

When the repository is cloned enter the directory to set up easy access to upstream changes:

```
$ cd faust
$ git remote add upstream git://github.com/robinhood/faust.git
$ git fetch upstream
```

If you need to pull in new changes from upstream you should always use the `--rebase` option to `git pull`:

```
$ git pull --rebase upstream master
```

With this option you don't clutter the history with merging commit notes. See Rebasing merge commits in git. If you want to learn more about rebasing see the Rebase section in the GitHub guides.

Start Developing

To start developing Faust you should install the requirements and setup the development environment so that Python uses the Faust development directory.

To do so run:

```
$ make develop
```

If you want to install requirements manually you should at least install the git pre-commit hooks (the `make develop` command above automatically runs this as well):

```
$ make hooks
```

If you also want to install C extensions, including the RocksDB bindings then you can use *make cdevelop* instead of *make develop*:

```
$ make cdevelop
```

Note: If you need to work on a different branch than the one git calls `master`, you can fetch and checkout a remote branch like this:

```
$ git checkout --track -b 2.0-devel origin/2.0-devel
```

Running the test suite

To run the Faust test suite you need to install a few dependencies. A complete list of the dependencies needed are located in `requirements/test.txt`.

Both the stable and the development version have testing related dependencies, so install these:

```
$ pip install -U -r requirements/test.txt
$ pip install -U -r requirements/default.txt
```

After installing the dependencies required, you can now execute the test suite by calling py.test <pytest:

```
$ py.test
```

This will run the unit tests, functional tests and doc example tests, but not integration tests or stress tests.

Some useful options to **`py.test`** are:

- `-x`

  Stop running the tests at the first test that fails.

- `-s`

  Don't capture output

- `-v`

  Run with verbose output.

If you want to run the tests for a single test file only you can do so like this:

```
$ py.test t/unit/test_app.py
```

Creating pull requests

When your feature/bugfix is complete you may want to submit a pull requests so that it can be reviewed by the maintainers.

Creating pull requests is easy, and also let you track the progress of your contribution. Read the Pull Requests section in the GitHub Guide to learn how this is done.

You can also attach pull requests to existing issues by following the steps outlined here: http://bit.ly/koJoso

Running the tests on all supported Python versions

There's a tox configuration file in the top directory of the distribution.

To run the tests for all supported Python versions simply execute:

```
$ tox
```

Use the `tox -e` option if you only want to test specific Python versions:

```
$ tox -e 2.7
```

Building the documentation

To build the documentation you need to install the dependencies listed in `requirements/docs.txt`:

```
$ pip install -U -r requirements/docs.txt
```

After these dependencies are installed you should be able to build the docs by running:

```
$ cd docs
$ rm -rf _build
$ make html
```

Make sure there are no errors or warnings in the build output. After building succeeds the documentation is available at `_build/html`.

Verifying your contribution

To use these tools you need to install a few dependencies. These dependencies can be found in `requirements/dist.txt`.

Installing the dependencies:

```
$ pip install -U -r requirements/dist.txt
```

pyflakes & PEP-8

To ensure that your changes conform to **PEP 8** and to run pyflakes execute:

```
$ make flakecheck
```

To not return a negative exit code when this command fails use the `flakes` target instead:

```
$ make flakes
```

API reference

To make sure that all modules have a corresponding section in the API reference please execute:

```
$ make apicheck
```

If files are missing you can add them by copying an existing reference file.

If the module is internal it should be part of the internal reference located in `docs/internals/reference/`. If the module is public it should be located in `docs/reference/`.

For example if reference is missing for the module `faust.worker.awesome` and this module is considered part of the public API, use the following steps:

Use an existing file as a template:

```
$ cd docs/reference/
$ cp faust.schedules.rst faust.worker.awesome.rst
```

Edit the file using your favorite editor:

```
$ vim faust.worker.awesome.rst

    # change every occurrence of ``faust.schedules`` to
    # ``faust.worker.awesome``
```

Edit the index using your favorite editor:

```
$ vim index.rst

    # Add ``faust.worker.awesome`` to the index.
```

Commit your changes:

```
# Add the file to git
$ git add faust.worker.awesome.rst
$ git add index.rst
$ git commit faust.worker.awesome.rst index.rst \
    -m "Adds reference for faust.worker.awesome"
```

Configuration Reference

To make sure that all settings have a corresponding section in the configuration reference, please execute:

```
$ make configcheck
```

If settings are missing from there an error is produced, and you can proceed by documenting the settings in `docs/userguide/settings.rst`.

1.8.8 Coding Style

You should probably be able to pick up the coding style from surrounding code, but it is a good idea to be aware of the following conventions.

- We use static types and the mypy type checker to verify them.

  Python code must import these static types when using them, so to keep static types lightweight we define interfaces for classes in `faust/types/`.

  For example for the `fauts.App` class, there is a corresponding *`faust.types.app.AppT`*; for *`faust.Channel`* there is a *`faust.types.channels.ChannelT`* and similarly for most other classes in the library.

  We suffer some duplication because of this, but it keeps static typing imports fast and reduces the need for recursive imports.

  In some cases recursive imports still happen, in that case you can "trick" the type checker into importing it, while regular Python does not:

  ```
  if typing.TYPE_CHECKING:
      from faust.app import App as _App
  else:
      class _App: ...  # noqa
  ```

  Note how we prefix the symbol with underscore to make sure anybody reading the code will think twice before using it.

- All Python code must follow the **PEP 8** guidelines.

pep8 is a utility you can use to verify that your code is following the conventions.

- Docstrings must follow the **PEP 257** conventions, and use the following style.

  Do this:

  ```
  def method(self, arg: str) -> None:
      """Short description.

      More details.

      """
  ```

  or:

  ```
  def method(self, arg: str) -> None:
      """Short description."""
  ```

  but not this:

```
def method(self, arg: str) -> None:
    """
    Short description.
    """
```

- Lines shouldn't exceed 78 columns.

  You can enforce this in **vim** by setting the `textwidth` option:

```
set textwidth=78
```

  If adhering to this limit makes the code less readable, you have one more character to go on. This means 78 is a soft limit, and 79 is the hard limit :)

- Import order

  - Python standard library
  - Third-party packages.
  - Other modules from the current package.

  or in case of code using Django:

  - Python standard library (*import xxx*)
  - Third-party packages.
  - Django packages.
  - Other modules from the current package.

  Within these sections the imports should be sorted by module name.

  Example:

```
import threading
import time
from collections import deque
from Queue import Queue, Empty

from .platforms import Pidfile
from .five import zip_longest, items, range
from .utils.time import maybe_timedelta
```

- Wild-card imports must not be used (*from xxx import **).

1.8.9 Contributing features requiring additional libraries

Some features like a new result backend may require additional libraries that the user must install.

We use setuptools *extra_requires* for this, and all new optional features that require third-party libraries must be added.

1) Add a new requirements file in *requirements/extras*

   For the RocksDB store this is `requirements/extras/rocksdb.txt`, and the file looks like this:

```
python-rocksdb
```

   These are pip requirement files so you can have version specifiers and multiple packages are separated by newline. A more complex example could be:

```
# python-rocksdb 2.0 breaks Foo
python-rocksdb>=1.0,<2.0
thrift
```

2) Modify `setup.py`

After the requirements file is added you need to add it as an option to `setup.py` in the `EXTENSIONS` section:

```
EXTENSIONS = {
    'debug',
    'fast',
    'rocksdb',
    'uvloop',
}
```

3) Document the new feature in `docs/includes/installation.txt`

You must add your feature to the list in the bundles section of `docs/includes/installation.txt`.

After you've made changes to this file you need to render the distro `README` file:

```
$ pip install -U requirements/dist.txt
$ make readme
```

1.8.10 Contacts

This is a list of people that can be contacted for questions regarding the official git repositories, PyPI packages Read the Docs pages.

If the issue isn't an emergency then it's better to *report an issue*.

Committers

Ask Solem

github https://github.com/ask

twitter http://twitter.com/#!/asksol

Vineet Goel

github https://github.com/vineet-rh

twitter https://twitter.com/#!/vineetik

Arpan Shah

github https://github.com/arpanshah29

1.8.11 Packages

`Faust`

git https://github.com/robinhood/faust

CI http://travis-ci.org/#!/robinhood/faust

Windows-CI https://ci.appveyor.com/project/ask/faust

PyPI faust

docs https://faust.readthedocs.io

`Mode`

git https://github.com/ask/mode

CI http://travis-ci.org/#!/ask/mode

Windows-CI https://ci.appveyor.com/project/ask/mode

PyPI Mode

docs http://mode.readthedocs.io/

1.8.12 Release Procedure

Updating the version number

The version number must be updated two places:

- `faust/__init__.py`
- `docs/include/introduction.txt`

After you have changed these files you must render the `README` files. There's a script to convert sphinx syntax to generic reStructured Text syntax, and the make target *readme* does this for you:

```
$ make readme
```

Now commit the changes:

```
$ git commit -a -m "Bumps version to X.Y.Z"
```

and make a new version tag:

```
$ git tag vX.Y.Z
$ git push --tags
```

Releasing

Commands to make a new public stable release:

```
$ make distcheck  # checks pep8, autodoc index, runs tests and more
$ make dist  # NOTE: Runs git clean -xdf and removes files not in the repo.
$ python setup.py sdist upload --sign --identity='Celery Security Team'
$ python setup.py bdist_wheel upload --sign --identity='Celery Security Team'
```

If this is a new release series then you also need to do the following:

- **Go to the Read The Docs management interface at:** http://readthedocs.org/projects/faust/?fromdocs=faust
- Enter "Edit project"

  Change default branch to the branch of this series, for example, use the `1.0` branch for the 1.0 series.
- Also add the previous version under the "versions" tab.

1.9 Developer Guide

Release 1.9

Date Jan 09, 2020

1.9.1 Contributors Guide to the Code

Module Overview

faust.app Defines the Faust application: configuration, sending messages, etc.

faust.cli Command-line interface.

faust.exceptions All custom exceptions are defined in this module.

faust.models Models describe how message keys and values are serialized/deserialized.

faust.sensors Sensors record statistics from a running Faust application.

faust.serializers Serialization using JSON, and codecs for encoding.

faust.stores Table storage: in-memory, RocksDB, etc.

faust.streams Stream and table implementation.

faust.topics Creating topic descriptions, and tools related to topics.

faust.transport Message transport implementations, e.g. aiokafka.

faust.types Public interface for static typing.

faust.utils Utilities. Note: This package is not allowed to import from the top-level package.

faust.web Web abstractions and web applications served by the Faust web server.

faust.windows Windowing strategies.

faust.worker Deployment helper for faust applications: signal handling, graceful shutdown, etc.

Services

Everything in Faust that can be started/stopped and restarted, is a `Service`.

Services can start other services, but they can also start `asyncio.Task` via *self.add_future*. These dependencies will be started/stopped/restarted with the service.

Worker

The worker can be used to start a Faust application, and performs tasks like setting up logging, installs signal handlers and debugging tools etc.

App

The app configures the Faust instance, and is the entry point for just about everything that happens in a Faust instance. Consuming/Producing messages, starting streams and agents, etc.

The app is usually started by `Worker`, but can also be started alone if less operating system interaction is wanted, like if you want to embed Faust in an application that already sets up signal handling and logging.

Monitor

The monitor is a feature-complete sensor that collects statistics about the running instance. The monitor data can be exposed by the web server.

Producer

The producer is used to publish messages to Kafka topics, and is started whenever necessary. The App will always starts this when a Faust instance is starting, in anticipation of messages to be produced.

Consumer

The Consumer is responsible for consuming messages from Kafka topics, to be delivered to the streams. It does not actually fetch messages (the `Fetcher` services does that), but it handles everything to do with consumption, like managing topic subscriptions etc.

Agent

Agents are also services, and any async function decorated using `@app.agent` will start with the app.

Conductor

The topic conductor manages topic subscriptions, and forward messages from the Kafka consumer to the streams.

`app.stream(topic)` will iterate over the topic: `aiter(topic)`. The conductor feeds messages into that iteration, so the stream receives messages in the topic:

```
async for event in stream(event async for event in topic)
```

TableManager

Manages tables, including recovery from changelog and caching table contents. The table manager also starts the tables themselves, and acts as a registry of tables in the Faust instance.

Table

Any user defined table.

Store

Every table has a separate store, the store describes how the table is stored in this instance. It could be stored in-memory (default), or as a RocksDB key/value database if the data set is too big to fit in memory.

Stream

These are individual streams, started after everything is set up.

Fetcher

The Fetcher is the service that actually retrieves messages from the kafka topic. The fetcher forwards these messages to the TopicManager, which in turns forwards it to Topic's and streams.

Web

This is a local web server started by the app (see `web_enable` setting).

1.9.2 Partition Assignor

Kafka Streams

Kafka Streams distributes work across multiple processes by using the consumer group protocol introduced in Kafka 0.9.0. Kafka elects one of the consumers in the consumer group to use its partition assignment strategy to assign partitions to the consumers in the group. The leader gets access to every client's subscriptions and assigns partitions accordingly.

Kafka Streams uses a sticky partition assignment strategy to minimize movement in the case of rebalancing. Further, it is also redundant in its partition assignment in the sense that it assigns some standby tasks to maintain state store replicas.

The StreamPartitionAssignor used by Kafka Streams works as follows:

1. Check all repartition source topics and use internal topic manager to make sure they have been created with the right number of partitions.
2. Using customized partition grouper (DefaultPartitionGrouper) to generate tasks along with their assigned partitions; also make sure that the task's corresponding changelog topics have been created with the right number of partitions.
3. Using StickyTaskAssignor to assign tasks to consumer clients.
   - Assign a task to a client which was running it previously. If there is no such client, assign a task to a client which has its valid local state.
   - A client may have more than one stream threads. The assignor tries to assign tasks to a client proportionally to the number of threads.
   - Try not to assign the same set of tasks to two different clients

   The assignment is done in one-pass. The result may not satisfy above all.
4. Within each client, tasks are assigned to consumer clients in round-robin manner.

Faust

Faust differs from Kafka Streams in some fundamental ways one of which is that a task in Faust differs from a task in Kafka Streams. Further, Faust doesn't have the concept of a pre-defined topology and subscribes to streams as and when required in the application.

As a result, the `PartitionAssignor` in Faust can get rid of steps one and two mentioned above and rely on the primitives repartitioning streams and creating changelog topics to create topics with the correct number of partitions based on the source topics.

We can largely simplify step three above since there is no concept of task as in Kafka Streams, i.e. we do not introspect the application topology to define a task that would be assigned to the clients. We simply need to make sure that the correct partitions are assigned to the clients and the client streams and processors should handle dealing with the co-partitioning while processing the streams and forwarding data between the different processors.

`PartitionGrouper`

This can be simplified immensely by grouping the same partition numbers onto the same clients for all topics with the same number of partitions. This way we can guarantee that co-partitioning for all topics requiring co-partitioning (ex: in the case of joins and aggregates) as long as the topics have the correct number of partitions (which we are making the processors implicitly guarantee).

`StickyAssignor`

With our simple *PartitionGrouper* we can use a `StickyPartitionAssignor` to assign partitions to the clients. However we need to explicitly handle standby assignments here. We use the `StickyPartitionAssignor` design approved in KIP-54 as the basis for our sticky assignor.

Concerns

With the above design we need to be careful around the following concerns:

- We need to assign a partition (where changelog) is involved to a client which contains a standby replica for the given topic/partition whenever possible. This can result in unbalanced assignment. We can fix this by evenly and randomly distributing standbys such that over the long term each rebalance will cause the partitions being re-assigned be evenly balanced across all clients.
- Network Partitions and other distributed systems failure cases - We delegate this to the Kafka protocol. The Kafka Consumer Protocol handles a lot of the failure conditions involved with the Consumer group leader election such as leader failures, node failures, etc. Network Partitions in Kafka are not handled here as those would result in bigger issues than consumer partition assignment issues.

1.10 History

This section contains historical change histories, for the latest version please visit *Changes*.

Release 1.9

Date Jan 09, 2020

1.10.1 Change history for Faust 1.8

This document contain change notes for bugfix releases in the Faust 1.8.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.8.1

release-date 2019-10-17 1:10 P.M PST

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 4.1.2.
- **Tables**: Fixed bug in table route decorator introduced in 1.8 (Issue #434).

  Fix contributed by Vikram Patki (@patkivikram).
- **Stream**: Now properly acks `None` values (tombstone messages).

  Fix contributed by Vikram Patki (@patkivikram).
- **Tables**: Fixed bug with `use_partitioner` when destination partitin is `0` (Issue #447).

  Fix contributed by Tobias Rauter (@trauter).
- **Consumer**: Livelock warning is now per TopicPartition.
- **Cython**: Add missing attribute in Cython class

  Fix contributed by Martin Maillard (@martinmaillard).
- **Distribution**: Removed broken gevent support (Issue #182, Issue #272).

  Removed all traces of gevent as the `aiogevent` project has been removed from PyPI.

  Contributed by Bryant Biggs (@bryantbiggs).
- Documentation fixes by:
  - Bryant Biggs (@bryantbiggs).

1.8.0

release-date 2019-09-27 4:05 P.M PST

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 4.1.0.
- **Tables**: New "global table" support (Issue #366).

  A global table is a table where all worker instances maintain a copy of the full table.

  This is useful for smaller tables that need to be shared across all instances.

  To define a new global table use `app.GlobalTable`:

```
global_table = app.GlobalTable('global_table_name')
```

Contributed by Artak Papikyan (@apapikyan).

- **Transports**: Fixed hanging when Kafka topics have gaps in source topic offset (Issue #401).

  This can happen when topics are compacted or similar, and Faust would previously hang when encountering offset gaps.

  Contributed by Andrei Tuppitcyn (@andr83).

- **Tables**: Fixed bug with crashing when key index enabled (Issue #414).

- **Streams**: Now properly handles exceptions in `group_by`.

  Contributed by Vikram Patki (@patkivikram).

- **Streams**: Fixed bug with `filter` not acking messages (Issue #391).

  Fix contributed by Martin Maillard (@martinmaillard).

- **Web**: Fixed typo in `NotFound` error.

  Fix contributed by Sanyam Satia (@ssatia).

- **Tables**: Added `use_partitioner` option for the ability to modify tables outside of streams (for example HTTP views).

  By default tables will use the partition number of a "source event" to write an entry to the changelog topic.

  This means you can safely modify tables in streams:

  ```
  async for key, value in stream.items():
     table[key] = value
  ```

  when the table is modified it will know what topic the source event comes from and use the same partition number.

  An alternative to this form of partitioning is to use the Kafka default partitioner on the key, and now you can use that strategy by enabling the `use_partitioner` option:

  ```
      table = app.Table('name', use_partitioner=True)

  You may also temporarily enable this option in any location
  by using ``table.clone(...)``:

  .. sourcecode:: python

      @app.page('/foo/{key}/')
      async def foo(web, request, key: str):
          table.clone(use_partitoner)[key] = 'bar'
  ```

- **Models**: Support for "schemas" that group key/value related settings together (Issue #315).

  This implements a single structure (Schema) that configures the `key_type`/`value_type`/`key_serializer`/`value_serializer` for a topic or agent:

  ```
      schema = faust.Schema(
          key_type=Point,
          value_type=Point,
          key_serializer='json',
          value_serializer='json',
      )
  ```

(continues on next page)

(continued from previous page)

```
    topic = app.topic('mytopic', schema=schema)

The benefit of having an abstraction a level above codecs
is that schemas can implement support for serialization formats
such as ProtocolBuffers, Apache Thrift and Avro.

The schema will also have access to the Kafka message headers,
necessary in some cases where serialization schema is specified
in headers.

.. seealso::

    :ref:`model-schemas` for more information.
```

- **Models**: Validation now supports optional fields (Issue #430).
- **Models**: Fixed support for `Optional` and field coercion (Issue #393).

  Fix contributed by Martin Maillard (@martinmaillard).

- **Models**: Manually calling `model.validate()` now also validates that the value is of the correct type (Issue #425).
- **Models**: Fields can now specify `input_name` and `output_name` to support fields named after Python reserved keywords.

  For example if the data you want to parse contains a field named `in`, this will not work since `in` is a reserved keyword.

  Using the new `input_name` feature you can rename the field to something else in Python, while still serializing/deserializing to the existing field:

```
from faust.models import Record
from faust.models.fields import StringField

class OpenAPIParameter(Record):
    location: str = StringField(default='query', input_name='in')
```

  `input_name` is the name of the field in serialized data, while `output_name` is what the field will be named when you serialize this model object:

```
>>> import json

>>> data = {'in': 'header'}
>>> parameter = OpenAPIParameter.loads(json.dumps(data))
>>> assert parameter.location == 'header'
>>> parameter.dumps(serialier='json')
'{"in": "header"}'
```

  Note:

  - The default value for `input_name` is the name of the field.
  - The default value for `output_name` is the value of `input_name`.

- **Models**: now have a `lazy_creation` class option to delay class initialization to a later time.

Field types are described using Python type annotations, and model fields can refer to other models, but not always are those models defined at the time when the class is defined.

Such as in this example:

```
class Foo(Record):
   bar: 'Bar'

class Bar(Record):
   foo: Foo
```

This example will result in an error, since trying to resolve the name `Bar` when the class `Foo` is created is impossible as that class does not exist yet.

In this case we can enable the `lazy_creation` option:

```
class Foo(Record, lazy_creation=True):
    bar: 'Bar'

class Bar(Record):
    foo: Foo

Foo.make_final()   # <-- 'Bar' is now defined so safe to create.
```

- **Transports**: Fixed type mismatch in aiokafka `timestamp_ms`

  Contributed by @ekerstens.

- **Models**: Added YAML serialization support.

  This requires the PyYAML library.

- **Sensors**: Added HTTP monitoring of status codes and latency.

- **App**: Added new *`Schema`* setting.

- **App**: Added new *`Event`* setting.

- **Channel**: A new `SerializedChannel` subclass can now be used to define new channel types that need to deserialize incoming messages.

- **Cython**: Added missing field declaration.

  Contributed by Victor Miroshnikov (@superduper)

- Documentation fixes by:
  - Adam Bannister (@AtomsForPeace).
  - Roman Imankulov (@imankulov).
  - Espen Albert (@EspenAlbert).
  - Alex Zeecka (@Zeecka).
  - Victor Noagbodji (@nvictor).
  - (@imankulov).
  - (@Zeecka).

1.10.2 Change history for Faust 1.7

This document contain change notes for bugfix releases in the Faust 1.7.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.7.3

release-date 2019-07-12 1:13 P.M PST

release-by Ask Solem (@ask)

- **Tables**: Fix for Issue #383 when using the Cython extension.

1.7.2

release-date 2019-07-12 12:00 P.M PST

release-by Ask Solem (@ask)

- **Tables**: Fixed memory leak/back pressure in changelog producer buffer (Issue #383)
- **Models**: Do not attempt to parse datetime when coerce/isodates disabled.

  Version 1.7 introduced a regression where datetimes were attempted to be parsed as ISO-8601 even with the `isodates` setting disabled.

  A regression test was added for this bug.

- **Models**: New `date_parser` option to change datetime parsing function.

  The default date parser supports ISO-8601 only. To support this format and many other formats (such as `'Sat Jan 12 00:44:36 +0000 2019'`) you can select to use python-dateutil as the parser.

  To change the date parsing function for a model globally:

```
from dateutil.parser import parse as parse_date

class Account(faust.Record, coerce=True, date_parser=parse_date):
    date_joined: datetime
```

  To change the date parsing function for a specific field:

```
from dateutil.parser import parse as parse_date
from faust.models.fields import DatetimeField

class Account(faust.Record, coerce=True):
    # datetime fields understands ISO 8601 only (default)
    date_joined: datetime

    # date_last_login comes from weird system with more human
    # readable dates ('Sat Jan 12 00:44:36 +0000 2019'),
    # the dateutil parser can handle many different date and time
    # formats.
    date_last_login: datetime = DatetimeField(date_parser=parse_date)
```

- **Models**: Adds `FieldDescriptor.exclude` to exclude field when serialized

  See *Excluding fields from representation* for more information.

- **Documentation**: improvements by…
  - Witek Bedyk (@witekest).
  - Josh Woodward (@jdw6359).

1.7.1

release-date 2019-07-09 2:36 P.M PST

release-by Ask Solem (@ask)

- **Stream**: Exactly once processing now include the app id in transactional ids.

  This was done to support running multiple apps on the same Kafka broker.

  Contributed by Cesar Pantoja (@CesarPantoja).

- **Web**: Fixed bug where sensor index should display when `debug` is enabled

  ---

  Tip: If you want to enable the sensor statistics endpoint in production, without enabling the `debug` setting, you can do so by adding the following code:

  ```
  app.web.blueprints.add('/stats/', 'faust.web.apps.stats:blueprint')
  ```

  Contributed by @tyong920

- **Transport**: The default value for `broker_request_timeout` is now 90 seconds (Issue #259)
- **Transport**: Raise error if `broker_session_timeout` is greater than `broker_request_timeout` (Closes #259)
- **Dependencies**: Now supports click 7.0 and later.
- **Dependencies**: `faust[debug]` now depends on aiomonitor 0.4.4 or later.
- **Models**: Field defined as `Optional[datetime]` now works with `coerce` and `isodates` settings.

  Previously a model would not recognize:

  ```
  class X(faust.Record, coerce=True):
      date: Optional[datetime]

  as a :class:`~faust.models.fields.DatetimeField` and when
  deserializing the field would end up as a string.

  It's now properly converted to :class:`~datetime.datetime`.
  ```

- **RocksDB**: Adds `table_key_index_size` setting (Closes #372)
- **RocksDB**: Reraise original error if python-rocksdb cannot be imported.

  Thanks to Sohaib Farooqi.

- **Django**: Autodiscovery support now waits for Django to be fully setup.

  Contributed by Tomasz Nguyen (@swist).

- **Documentation** improvements by:
  - Witek Bedyk (@witekest).

1.7.0

release-date 2019-06-06 6:00 P.M PST

release-by Ask Solem (@ask)

Backward Incompatible Changes

- **Transports**: The in-memory transport has been removed (Issue #295).

  This transport was experimental and not working properly, so to avoid confusion we have removed it completely.

- **Stream**: The `Message.stream_meta` attribute has been removed.

  This was used to keep arbitrary state for sensors during processing of a message.

  If you by rare chance are relying on this attribute to exist, you must now initialize it before using it:

```
stream_meta = getattr(event.message, 'stream_meta', None)
if stream_meta is None:
    stream_meta = event.message.stream_meta = {}
```

News

- **Requirements**
  - Now depends on Mode 4.0.0.
  - Now depends on aiohttp 3.5.2 or later.

    Thanks to @CharAct3.

- **Documentation**: Documented a new deployment strategy to minimize rebalancing issues.

  See *Managing a cluster* for more information.

- **Models**: Implements model validation.

  Validation of fields can be enabled by using the `validation=True` class option:

```
import faust
from decimal import Decimal

class X(faust.Record, validation=True):
    name: str
    amount: Decimal
```

  When validation is enabled, the model will validate that the fields values are of the correct type.

  Fields can now also have advanced validation options, and you enable these by writing explicit field descriptors:

```
import faust
from decimal import Decimal
from faust.models.fields import DecimalField, StringField

class X(faust.Record, validation=True):
    name: str = StringField(max_length=30)
    amount: Decimal = DecimalField(min_value=10.0, max_value=1000.0)
```

If you want to run validation manually, you can do so by keeping `validation=False` on the class, but calling `model.is_valid()`:

```
if not model.is_valid():
    print(model.validation_errors)
```

- **Models**: Implements generic coercion support.

  This new feature replaces the `isodates=True`/`decimals=True` options and can be enabled by passing `coerce=True`:

  ```
  class Account(faust.Record, coerce=True):
      name: str
      login_times: List[datetime]
  ```

- **Testing**: New experimental `livecheck` production testing API.

  There is no documentation yet, but an example in `examples//livecheck.py`.

  This is a new API to do end-to-end testing directly in production.

- **Topic**: Adds new `topic.send_soon()` non-async method to buffer messages.

  This method can be used by any non-*async def* function to buffer up messages to be produced.

  It returns *Awaitable[RecordMetadata]*: a promise evaluated once the message is actually sent.

- **Stream**: New `Stream.filter` method added useful for filtering events before repartitioning a stream.

  See *filter() – Filter values to omit from stream.* for more information.

- **App**: New *broker_consumer*/*broker_producer* settings.

  These can now be used to configure individual transports for consuming and producing.

  The default value for both settings are taken from the *broker* setting.

  For example you can use aiokafka for the consumer, and confluent_kafka for the producer:

  ```
  app = faust.App(
      'id',
      broker_consumer='kafka://localhost:9092',
      broker_producer='confluent://localhost:9092',
  )
  ```

- **App**: New *broker_max_poll_interval* setting.

  Contributed by Miha Troha (@mihatroha).

- **App**: New *topic_disable_leader* setting disables the leader topic.

- **Table**: Table constructor now accepts `options` argument passed on to underlying RocksDB storage.

  This can be used to configure advanced RocksDB options, such as block size, cache size, etc.

  Contributed by Miha Troha (@mihatroha).

Fixes

- **Stream**: Fixes bug where non-finished event is acked (Issue #355).
- **Producer**: Exactly once: Support producing to non-transactional topics (Issue #339)
- **Agent**: Test: Fixed `asyncio.CancelledError` (Issue #322).
- **Cython**: Fixed issue with sensor state not being passed to `after`.
- **Tables**: Key index: now inherits configuration from source table (Issue #325)
- **App**: Fix list of strings for *`broker`* param in URL (Issue #330).

  Contributed by Nimish Telang (@nimish).
- **Table**: Fixed blocking behavior when populating tables.

  Symptom was warnings about timers waking up too late.
- **Documentation** Fixes by:
  - @evanderiel

Improvements

- **Documentation**: Rewrote fragmented documentation to be more concise.
- **Documentation improvements by**
  - Igor Mozharovsky (@seedofjoy)
  - Stephen Sorriaux (@StephenSorriaux)
  - Lifei Chen (@hustclf)

1.10.3 Change history for Faust 1.6

This document contain change notes for bugfix releases in the Faust 1.6.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.6.1

release-date 2019-05-07 2:00 P.M PST

release-by Ask Solem (@ask)

- **Web**: Fixes index page of web server by adding `collections.deque` support to our JSON serializer.

  Thanks to Brandon Ewing for detecting this issue.

1.6.0

release-date 2019-04-16 5:41 P.M PST

release-by Ask Solem (@ask)

This release has minor backward incompatible changes. that only affects those who are using custom sensors. See note below.

- **Requirements**:
  - Now depends on robinhood-aiokafka 1.0.3

    This version disables the "LeaveGroup" timeout added in 1.0.0, as it was causing problems.
- **Sensors**: `on_stream_event_in` now passes state to `on_stream_event_out`.

  This is backwards incompatible but fixes a rare race condition.

  Custom sensors that have to use stream_meta must be updated to use this state.
- **Sensors**: Added new sensor methods:
  - `on_rebalance_start(app)`

    Called when a new rebalance is starting.
  - `on_rebalance_return(app)`

    Called when the worker has returned data to Kafka.

    The next step of the rebalancing phase will be the table recovery process, but this happens in the background and rebalancing will be considered complete for this worker.
  - `on_rebalance_end(app)`

    Called when all tables are fully recovered and the worker is ready to start processing events in the stream.
- **Sensors**: The type of a sensor that returns/takes state is now Dict instead of a Mapping (as the state is mutable).
- **Monitor**: Optimized latency history cleanup.
- **Recovery**: Fixed bug with highwater returning `None`.
- **Tracing**: The `traced` decorator would return `None` for wrapped coroutines, but we now return the actual return value.
- **Tracing: Added tracing of aiokafka group coordinator processes** (rebalancing and find coordinator).

1.10.4 Change history for Faust 1.5

This document contain change notes for bugfix releases in the Faust 1.5.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.5.4

release-date 2019-04-9 2:09 P.M PST

release-by Ask Solem (@ask)

- New *producer_api_version* setting.

  This can be set to the value "0.10" to remove headers from all messages produced.

  Use this if you have downstream consumers that do not support the new Kafka protocol format yet.

- The *stream_recovery_delay* setting has been disabled by default.

  After rebalancing the worker will sleep a bit before starting recovery, the idea being that another recovery may be waiting just behind it so we wait a bit, but this has shown to be not as effective as intended.

- **Web**: Cache can now be configured to take headers into account.

  Create the cache manager for your blueprint with the `include_headers` argument:

  ```
  cache = blueprint.cache(timeout=300.0, include_headers=True)
  ```

  Contributed by Sanyam Satia (@ssatia).

1.5.3

release-date 2019-04-06 11:25 P.M PST

release-by Ask Solem (@ask)

- **Requirements**:
  - Now depends on robinhood-aiokafka 1.0.2

    This version disables the "LeaveGroup" timeout added in 1.0.0, as it was causing problems.

- **Documentation**: Fixed spelling.
- **Tests**: Fixed flaky regression test.

1.5.2

release-date 2019-03-28 11:00 A.M PST

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 3.1.1.
- Timers: Prevent drift + add some tiny drift.

  Thanks to Bob Haddleton (@bobh66).

- **App**: Autodiscovery now avoids importing `__main__.py` (Issue #324).

  Added regression test.

- The `stream_ack_exceptions` setting has been deprecated.

  It was not having any effect, and we have no current use for it.

- The `stream_ack_cancelled_tasks` setting has been deprecated.

It was not having any effect, and we have no current use for it.

- **App**: Autodiscovery failed to load when using `app.main()` in some cases (Issue #323).

  Added regression test.

- **Worker**: Fixed error during agent shutdown.
- **Monitor**: Monitor assignment latency + assignments completed/failed.

  Implemented in the default monitor, but also for statsd and datadog.

- **CLI**: The **`faust`** program had the wrong help description.
- **Docs**: Fixes typo in *`web_cors_options`* example.
- **App**: Do no wait for table recovery finished signal, if the app is not starting the recovery service.

1.5.1

release-date 2019-03-24 09:45 P.M PST

release-by Ask Solem (@ask)

- Fixed hanging in partition assignment introduced in Faust 1.5 (Issue #320).

  Contributed by Bob Haddleton (@bobh66).

1.5.0

release-date 2019-03-22 02:18 P.M PST

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on robinhood-aiokafka 1.0.1
  - Now depends on Mode 3.1.
- Exactly-Once semantics: New *`processing_guarantee`* setting.

  Experimental support for "exactly-once" semantics.

  This mode ensures tables and counts in tables/windows are consistent even as nodes in the cluster are abruptly terminated.

  To enable this mode set the *`processing_guarantee`* setting:

  ```
  App(processing_guarantee='exactly_once')
  ```

  Note: If you do enable "exactly_once" for an existing app, you must make sure all workers are running the latest version and possibly starting from a clean set of intermediate topics.

  You can accomplish this by bumping up the app version number:

  ```
  App(version=2, processing_guarantee='exactly_once')
  ```

  The new processing guarantee require a new version of the assignor protocol, for this reason a "exactly_once" worker will not work with older versions of Faust running in the same consumer group: so

to roll out this change you will have to stop all the workers, deploy the new version and only then restart the workers.

- New optimizations for stream processing and windows.

  If Cython is available during installation, Faust will be installed with compiled extensions.

  You can set the `NO_CYTHON` environment variable to disable the use of these extensions even if compiled.

- New *`topic_allow_declare`* setting.

  If disabled your faust worker instances will never actually declare topics.

  Use this if your Kafka administrator does not allow you to create topics.

- New *`ConsumerScheduler`* setting.

  This class can override how events are delivered to agents. The default will go round robin between both topics and partitions, to ensure all topic partitions get a chance of being processed.

  Contributed by Miha Troha (@miatroha).

- **Authentication**: Support for GSSAPI authentication.

  See documentation for the *`broker_credentials`* setting.

  Contributed by Julien Surloppe (@jsurloppe).

- **Authentication**: Support for SASL authentication.

  See documentation for the *`broker_credentials`* setting.

- New *`broker_credentials`* setting can also be used to configure SSL authentication.

- **Models**: Records can now use comparison operators.

  Comparison of models using the >, <, >= and <= operators now work similarly to `dataclasses`.

- **Models**: Now raise an error if non-default fields follows default fields.

  The following model will now raise an error:

```
class Account(faust.Record):
    name: str
    amount: int = 3
    userid: str
```

  This is because a non-default field is defined after a default field, and this would mess up argument ordering.

  To define the model without error, make sure you move default fields below any non-default fields:

```
class Account(faust.Record):
    name: str
    userid: str
    amount: int = 3
```

Note: Remember that when adding fields to an already existing model you should always add new fields as optional fields.

This will help your application stay backward compatible.

- **App**: Sending messages API now supports a `headers` argument.

  When sending messages you can now attach arbitrary headers as a dict, or list of tuples; where the values are bytes:

  ```
  await topic.send(key=key, value=value, headers={'x': b'foo'})
  ```

  Supported transports

  Headers are currently only supported by the default aiokafka transport, and requires Kafka server 0.11 and later.

- **Agent**: RPC operations can now take advantage of message headers.

  The default way to attach metadata to values, such as the reply-to address and the correlation id, is to wrap the value in an envelope.

  With headers support now landed we can use message headers for this:

  ```
  @app.agent(use_reply_headers=True)
  async def x(stream):
      async for item in stream:
          yield item ** 2
  ```

  Faust will be using headers by default in version 2.0.

- **App**: Sending messages API now supports a `timestamp` argument (Issue #276).

  When sending messages you can now specify the timestamp of the message:

  ```
  await topic.send(key=key, value=value, timestamp=custom_timestamp)
  ```

  If no timestamp is provided the current time will be used (`time.time()`).

  Contributed by Miha Troha (@mihatroha).

- **App**: New *`consumer_auto_offset_reset`* setting (Issue #267).

  Contributed by Ryan Whitten (@rwhitten577).

- **Stream**: `group_by` repartitioned topic name now includes the agent name (Issue #284).

- **App**: Web server is no longer running in a separate thread by default.

  Running the web server in a separate thread is beneficial as it will not be affected by back pressure in the main thread event loop, but it also makes programming harder when it cannot share the loop of the parent.

  If you want to run the web server in a separate thread, use the new *`web_in_thread`* setting.

- **App**: New *`web_in_thread`* controls separate thread for web server.

- **App**: New *`logging_config`* setting.

- **App**: Autodiscovery now ignores modules matching "*test*" (Issue #242).

  Contributed by Chris Seto (@chrisseto).

- **Transport**: aiokafka transport now supports headers when using Kafka server versions 0.11 and later.

- **Tables**: New flags can be used to check if actives/standbys are up to date.

  - `app.tables.actives_ready`

Set to `True` when tables have synced all active partitions.

- `app.tables.standbys_ready`

  Set to `True` when standby partitions are up-to-date.

- **RocksDB**: Now crash with `ConsistencyError` if the persisted offset is greater than the current highwater.

  This means the changelog topic has been modified in Kafka and the recorded offset no longer exists. We crash as we believe this require human intervention, but should some projects have less strict durability requirements we may make this an option.

- **RocksDB**: `len(table)` now only counts databases for active partitions (Issue #270).
- **Agent**: Fixes crash when worker assigned no partitions and having the `isolated_partitions` flag enabled (Issue #181).
- **Table**: Fixes `KeyError` crash for already removed key.
- **Table**: WindowRange is no longer a `NamedTuple`.

  This will make it easier to avoid hashing mistakes such that window ranges are never represented as both normal tuple and named tuple variants in the table.

- **Transports**: Adds experimental `confluent://` transport.

  This transport uses the confluent-kafka client.

  It is not feature complete, and notably is missing sticky partition assignment so you should not use this transport for tables.

  > **Warning:** The `confluent://` transport is not recommended for production use at this time as it has several limitations.

- **Stream**: Fixed deadlock when using `Stream.take` to buffer events (Issue #262).

  Contributed by Nimi Wariboko Jr (@nemosupremo).

- **Web**: Views can now define `options` method to implement a handler for the HTTP `OPTIONS` method. (Issue #304)

  Contributed by Perk Lim (@perklun).

- **Stream**: Fixed acking behavior of `Stream.take` (Issue #266).

  When `take` is buffering the events should be acked after processing the buffer is complete, instead it was acking when adding into the buffer.

  Fix contributed by Amit Ripshtos (@amitripshtos).

- **Transport: Aiokafka was not limiting how many messages to read in** a fetch request (Issue #292).

  Fix contributed by Miha Troha (@mihatroha).

- **Typing**: Added type stubs for `faust.web.Request`.
- **Typing**: Fixed type stubs for `@app.agent` decorator.
- **Web**: Added support for Cross-Resource Origin Sharing headers (CORS).

  See new *`web_cors_options`* setting.

- **Debugging: Added OpenTracing hooks to streams/tasks/timers/Crontabs** and rebalancing process.

To enable you have to define a custom `Tracer` class that will record and publish the traces to systems such as Jeager or Zipkin.

This class needs to have a `.trace(name, **extra_context)` context manager:

```
from typing import Any, Dict,
import opentracing
from opentracing.ext.tags import SAMPLING_PRIORITY

class FaustTracer:
    _tracers: Dict[str, opentracing.Tracer]
    _default_tracer: opentracing.Tracer = None

    def __init__(self) -> None:
        self._tracers = {}

    @cached_property
    def default_tracer(self) -> opentracing.Tracer:
        if self._default_tracer is None:
            self._default_tracer = self.get_tracer('APP_NAME')

    def trace(self, name: str,
              sample_rate: float = None,
              **extra_context: Any) -> opentracing.Span:
            span = self.default_tracer.start_span(
            operation_name=name,
            tags=extra_context,
        )

        if sample_rate is not None:
            priority = 1 if random.uniform(0, 1) < sample_rate else 0
            span.set_tag(SAMPLING_PRIORITY, priority)
        return span

    def get_tracer(self, service_name: str) -> opentracing.Tracer:
        tracer = self._tracers.get(service_name)
        if tracer is None:
            tracer = self._tracers[service_name] = CREATE_
↪TRACER(service_name)
        return tracer._tracer
```

After implementing the interface you need to set the `app.tracer` attribute:

```
app = faust.App(...)
app.tracer = FaustTracer()
```

That's it! Now traces will go through your custom tracing implementation.

- **CLI**: Commands `--help` output now always show the default for every parameter.
- **Channels**: Fixed bug in `channel.send` that caused a memory leak.

  This bug was not present when using `app.topic()`.
- **Documentation**: Improvements by:
  - Amit Rip (@amitripshtos).
  - Sebastian Roll (@SebastianRoll).
  - Mousse (@zibuyu1995).

  - Zhanzhao (Deo) Liang (@DeoLeung).

- **Testing**:
  - 99% total unit test coverage
  - New script to verify documentation defaults are up to date are run for every git commit.

1.10.5 Change history for Faust 1.4

This document contain change notes for bugfix releases in the Faust 1.4.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.4.9

release-date 2019-03-14 04:00 P.M PST

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 3.0.10.
- `max_poll_records` accidentally set to 500 by default.

  The setting has been reverted to its documented default of `None`. This resulted in a 20x performance improvement.
- **CLI**: Now correctly returns non-zero exitcode when exception raised inside `@app.command`.
- **CLI**: Option `--no_color` renamed to `--no-color` to be consistent with other options.

  This change is backwards compatible and `--no_color` will continue to work.
- **CLI**: The `model x` command used "default*" as the field name for default value.

```
$ python examples/withdrawals.py --json model Withdrawal | python -m json.
↪tool
[
    {
        "field": "user",
        "type": "str",
        "default*": "*"
    },
    {
        "field": "country",
        "type": "str",
        "default*": "*"
    },
    {
        "field": "amount",
        "type": "float",
        "default*": "*"
    },
    {
        "field": "date",
        "type": "datetime",
        "default*": "None"
```

(continues on next page)

(continued from previous page)

```
    }
]
```

This now gives "default" without the extraneous star.

- **App**: Can now override the settings class used.

  This means you can now easily extend your app with custom settings:

```
import faust

class MySettings(faust.Settings):
    foobar: int

    def __init__(self, id: str, *, foobar: int = 0, **kwargs) -> None:
        super().__init__(id, **kwargs)
        self.foobar = foobar

class App(faust.App):
    Settings = MySettings

app = App('id', foobar=3)
print(app.conf.foobar)
```

1.4.8

release-date 2019-03-11 05:30 P.M PDT

release-by Ask Solem (@ask)

- **Tables**: Recovery would hang when changelog have `committed_offset == 0`.

  Added this test to our manual testing procedure.

1.4.7

release-date 2019-03-08 02:21 P.M PDT

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 3.0.9.
- **Tables**: Read offset not always updated after seek caused recovery to hang.
- **Consumer**: Fix to make sure fetch requests will not block method queue.
- **App**: Fixed deadlock in rebalancing.
- **Web**: Views can now define `options` method to implement a handler for the HTTP `OPTIONS` method. (Issue #304)

  Contributed by Perk Lim (@perklun).
- **Web**: Can now pass headers to HTTP responses.

1.4.6

release-date 2019-01-29 01:52 P.M PDT

release-by Ask Solem (@ask)

- **App**: Better support for custom boot strategies by having the app start without waiting for recovery when no tables started.
- **Docs: Fixed doc build after intersphinx** URL https://click.palletsprojects.com/en/latest no longer works.

1.4.5

release-date 2019-01-18 02:15 P.M PDT

release-by Ask Solem (@ask)

- Fixed typo in 1.4.4 release (on_recovery_set_flags -> on_rebalance_start).

1.4.4

release-date 2019-01-18 01:10 P.M PDT

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 3.0.7.
- **App**: App now starts even if there are no agents defined.
- **Table**: Added new flags to detect if actives/standbys are ready.
  - `app.tables.actives_ready`

    Set to `True` when active tables are recovered from and are ready to use.
  - `app.tables.standbys_ready`

    Set to `True` when standbys are up to date after recovery.

1.4.3

release-date 2019-01-14 03:01 P.M PDT

release-by Ask Solem (@ask)

- **Requirements**
  - Require series 0.4.x of robinhood-aiokafka.
    - Recently version 0.5.0 was released but this has not been tested in production yet, so we have pinned Faust 1.4.x to aiokafka 0.4.x. For more information see Issue #277.
  - Test requirements now depends on pytest greater than 3.6.

    Contributed by Michael Seifert (@seifertm).
- **Documentation improvements by**:
  - Allison Wang (@allisonwang).
  - Thibault Serot (@thibserot).

  - @oucb.

- **CI**: Added CPython 3.7.2 and 3.6.8 to Travis CI build matrix.

1.4.2

release-date 2018-12-19 12:49 P.M PDT

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 3.0.5.

    Fixed compatibility with colorlog, thanks to Ryan Whitten (@rwhitten577).
  - Now compatible with yarl 1.3.x.
- **Agent**: Allow `yield` in agents that use `Stream.take` (Issue #237).
- **App: Fixed error "future for different event loop" when web views** send messages to Kafka at startup.
- **Table**: Table views now return HTTP 503 status code during startup when table routing information not available.
- **App**: New `App.BootStrategy` class now decides what services are started when starting the app.
- Documentation fixes by:
  - Robert Krzyzanowski (@robertzk).

1.4.1

release-date 2018-12-10 4:49 P.M PDT

release-by Ask Solem (@ask)

- **Web**: Disable aiohttp access logs for performance.

1.4.0

release-date 2018-12-07 4:29 P.M PDT

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 3.0.
- **Worker**: The Kafka consumer is now running in a separate thread.

  The Kafka heartbeat background coroutine sends heartbeats every 3.0 seconds, and if those are missed rebalancing occurs.

  This patch moves the aiokafka library inside a separate thread, this way it can send responsive heartbeats and operate even when agents call blocking functions such as `time.sleep(60)` for every event.
- **Table**: Experimental support for tables where values are sets.

  The new `app.SetTable` constructor creates a table where values are sets. Example uses include keeping track of users at a location: `table[location].add(user_id)`.

  Supports all set operations: `add`, `discard`, `intersection`, `union`, `symmetric_difference`, `difference`, etc.

Sets are kept in memory for fast operation, and this way we also avoid the overhead of constantly serializing/deserializing the data to RocksDB. Instead we periodically flush changes to RocksDB, and populate the sets from disk at worker startup/table recovery.

- **App**: Adds support for Crontab tasks.

  You can now define periodic tasks using Cron-syntax:

```
@app.crontab('*/1 * * * *', on_leader=True)
async def publish_every_minute():
    print('-- We should send a message every minute --')
    print(f'Sending message at: {datetime.now()}')
    msg = Model(random=round(random(), 2))
    await tz_unaware_topic.send(value=msg).
```

  See *Cron Jobs* for more information.

  Contributed by Omar Rayward (@omarrayward).

- **App**: Providing multiple URLs to the `broker` setting now works as expected.

  To facilitate this change `app.conf.broker` is now `List[URL]` instead of a single `URL`.

- **App**: New `timezone` setting.

  This setting is currently used as the default timezone for Crontab tasks.

- **App**: New `broker_request_timeout` setting.

  Contributed by Martin Maillard (@martinmaillard).

- **App**: New `broker_max_poll_records` setting.

  Contributed by Alexander Oberegger (@aoberegg).

- **App**: New `consumer_max_fetch_size` setting.

  Contributed by Matthew Stump (@mstump).

- **App**: New `producer_request_timeout` setting.

  Controls when producer batch requests expire, and when we give up sending batches as producer requests fail.

  This setting has been increased to 20 minutes by default.

- **Web**: aiohttp driver now uses `AppRunner` to start the web server.

  Contributed by Mattias Karlsson (@stevespark).

- **Agent**: Fixed RPC example (Issue #155).

  Contributed by Mattias Karlsson (@stevespark).

- **Table**: Added support for iterating over windowed tables.

  See *Iterating over keys/values/items in a windowed table.*.

  This requires us to keep a second table for the key index, so support for windowed table iteration requires you to set a `use_index=True` setting for the table:

```
windowed_table = app.Table(
    'name',
    default=int,
).hopping(10, 5, expires=timedelta(minutes=10), key_index=True)
```

  After enabling the `key_index=True` setting you may iterate over keys/items/values in the table:

```
for key in windowed_table.keys():
    print(key)

for key, value in windowed_table.items():
    print(key, value)

for value in windowed_table.values():
    print(key, value)
```

The `items` and `values` views can also select time-relative iteration:

```
for key, value in windowed_table.items().delta(30):
    print(key, value)
for key, value in windowed_table.items().now():
    print(key, value)
for key, value in windowed_table.items().current():
    print(key, value)
```

- **Table: Now raises error if source topic has mismatching** number of partitions with changelog topic. (Issue #137).
- **Table**: Allow using raw serializer in tables.

  You can now control the serialization format for changelog tables, using the `key_serializer` and `value_serializer` keyword arguments to `app.Table(...)`.

  Contributed by Matthias Wutte (@wuttem).
- **Worker**: Fixed spinner output at shutdown.
- **Models**: `isodates` option now correctly parses timezones without separator such as *-0500*.
- **Testing**: Calling `AgentTestWrapper.put` now propagates exceptions raised in the agent.
- **App**: Default value for *`stream_recovery_delay`* is now 3.0 seconds.
- **CLI**: New command "clean_versions" used to delete old version directories (Issue #68).
- **Web**: Added view decorators: `takes_model` and `gives_model`.

1.10.6 Change history for Faust 1.3

This document contain change notes for bugfix releases in the Faust 1.3.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.3.2

release-date 2018-11-19 1:11 P.M PST

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 2.0.4.
- Fixed crash in `perform_seek` when worker was not assigned any partitions.
- Fixed missing `await` in `Consumer.wait_empty`.

- Fixed hang after rebalance when not using tables.

1.3.1

release-date 2018-11-15 4:12 P.M PST

release-by Ask Solem (@ask)

- **Tables**: Fixed problem with table recovery hanging on changelog topics having only a single entry.

1.3.0

release-date 2018-11-08 4:49 P.M PST

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 2.0.3.
  - Now depends on `robinhood-aiokafka` 1.4.19
- **App**: Refactored rebalancing and table recovery (Issue #185).

  This optimizes the rebalancing callbacks for greater stability.

  Table recovery was completely rewritten to do as little as possible during actual rebalance. This should increase stability and reduce the chance of rebalancing loops.

  We no longer attempt to cancel recovery during rebalance, so this should also fix problems with hanging during recovery.
- **App**: Adds new *`stream_recovery_delay`* setting.

  In this version we are experimenting with sleeping for 10.0 seconds after rebalance, to allow for more nodes to join/leave before resuming the streams.

  This adds some startup delay, but is in general unnoticeable in production.
- **Windowing**: Fixed several edge cases in windowed tables.

  Fix contributed by Omar Rayward (@omarrayward).
- **App**: Skip table recovery on rebalance when no tables defined.
- **RocksDB**: Iterating over table keys/items/values now skips standby partitions.
- **RocksDB**: Fixed issue with having "." in table names (Issue #184).
- **App**: Allow *`broker`* URL setting without scheme.

  The default scheme for an URL like "localhost:9092" is `kafka://`.
- **App**: Adds `App.on_rebalance_complete` signal.
- **App**: Adds `App.on_before_shutdown` signal.
- **Misc**: Support for Python 3.8 by importing from *collections.abc*.
- **Misc**: Got rid of aiohttp deprecation warnings.
- **Documentation and examples**: Improvements contributed by:
  - Martin Maillard (@martinmaillard).
  - Omar Rayward (@omarrayward).

1.10.7 Change history for Faust 1.2

This document contain change notes for bugfix releases in the Faust 1.2.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.2.2

- **Requirements**
  - Now depends on `aiocontextvars` 0.1.x.

    The new 0.2 version is backwards incompatible and breaks Faust.
- **Settings**: Increases default *`broker_session_timeout`* to 60.0 seconds.
- **Tables**: Fixes use of windowed tables when using `simplejson`.

  This change makes sure `simplejson` serializes `typing.NamedTuple` as lists, and not dictionaries.

  Fix contributed by Omar Rayward (@omarrayward).
- **Tables**: `windowed_table[key].now()` works outside of stream iteration.

  Fix contributed by Omar Rayward (@omarrayward).
- **Examples**: New Kubernetes example.

  Contributed by Omar Rayward (@omarrayward).
- **Misc**: Fixes `DeprecationWarning` for `asyncio.current_task`.
- **Typing**: Type checks now compatible with mypy 0.641.
- Documentation and examples fixes contributed by
  - Fabian Neumann (@hellp)
  - Omar Rayward (@omarrayward)

1.2.1

release-date 2018-10-08 5:00 P.M PDT

release-by Ask Solem (@ask)

- **Worker**: Fixed crash introduced in 1.2.0 if no `--loglevel` argument present.
- **Web**: The aiohttp driver now exposes `app.web.web_app` attribute.

  This will be the `aiohttp.web_app.Application` instance used.
- **Documentation**: Fixed markup typo in the settings section of the *User Guide* (Issue #177).

  Contributed by Denis Kataev (@kataev).

1.2.0

release-date 2018-10-05 5:23 P.M PDT

release-by Ask Solem (@ask).

Fixes

- **CLI: All commands, including user-defined, now wait for producer to** be fully stopped before shutting down to make sure buffers are flushed (Issue #172).
- **Table**: Delete event in changelog would crash app on table restore (Issue #175)
- **App: Channels and topics now take default** `key_serializer`/`value_serializer` from `key_type`/`value_type` when they are specified as models (Issue #173).

  This ensures support for custom codecs specified using the model `serializer` class keyword:

```
class X(faust.Record, serializer='msgpack'):
    x: int
    y: int
```

News

- **Requirements**
  - Now depends on Mode 1.18.1.
- **CLI**: Command-line improvements.
  - All subcommands are now executed by `mode.Worker`.

    This means all commands will have the same environment set up, including logging, signal handling, blocking detection support, and remote aiomonitor console support.
  - `faust worker` options moved to top level (built-in) options:
    - `--logfile`
    - `--loglevel`
    - `--console-port`
    - `--blocking-timeout`

    To be backwards compatible these options can now appear before and after the `faust worker` command on the command-line (but for all other commands they need to be specified before the command name):

```
$ ./examples/withdrawals.py -l info worker  # OK
$ ./examples/withdrawals.py worker -l info  # OK
$ ./examples/withdrawals.py -l info agents  # OK
$ ./examples/withdrawals.py agents -l info  # error!
```

  - If you want a long running background command that will run even after returning, use: `daemon=True`.

If enabled the program will not shut down until either the user hits `Control-c`, or the process is terminated by a signal:

```
@app.command(daemon=True)
async def foo():
    print('starting')
    # set up stuff
    return  # command will continue to run after return.
```

- **CLI**: New *`call_command()`* utility for testing.

  This can be used to safely call a command by name, given an argument list.

- **Producer**: New *`producer_partitioner`* setting (Issue #164)
- **Models**: Attempting to instantiate abstract model now raises an error (Issue #168).
- **App**: App will no longer raise if configuration accessed before being finalized.

  Instead there's a new *`AlreadyConfiguredWarning`* emitted when a configuration key that has been read is modified.

- **Distribution: Setuptools metadata now moved to `setup.py` to**

  keep in one location.

  This also helps the README banner icons show the correct information.

  Contributed by Bryant Biggs (@bryantbiggs)

- Documentation and examples improvements by
  - Denis Kataev (@kataev).

Web Improvements

Note: `faust.web` is a small web abstraction used by Faust projects.

It is kept separate and is decoupled from stream processing so in the future we can move it to a separate package if necessary.

You can safely disable the web server component of any Faust worker by passing the `--without-web` option.

- **Web: Users can now disable the web server from the faust worker** (Issue #167).

  Either by passing *`faust worker --without-web`* on the command-line, or by using the new `web_enable` setting.

- **Web**: Blueprints can now be added to apps by using strings

  Example:

```
app = faust.App('name')

app.web.blueprints.add('/users/', 'proj.users.views:blueprint')
```

- **Web**: Web server can now serve using Unix domain sockets.

  The `--web-transport` argument to **`faust worker`**, and the *`web_transport`* setting was added for this purpose.

  Serve HTTP over Unix domain socket:

```
faust -A app -l info worker --web-transport=unix:///tmp/faustweb.sock
```

- **Web: Web server is now started by the *App***

  faust.Worker.

  This makes it easier to access web-related functionality from the app. For example to get the URL for a view by name, you can now use `app.web` to do so after registering a blueprint:

```
app.web.url_for('user:detail', user_id=3)
```

- New *web* allows you to specify web framework by URL.

  Default, and only supported web driver is currently `aiohttp://`.

- **View**: A view can now define `__post_init__`, just like dataclasses/Faust models can.

  This is useful for when you don't want to deal with all the work involved in overriding `__init__`:

```
@blueprint.route('/', name='list')
class UserListView(web.View):

    def __post_init__(self):
        self.something = True

    async def get(self, request, response):
        if self.something:
            ...
```

- **aiohttp Driver: `json()` response method now uses the Faust json** serializer for automatic support of `__json__` callbacks.

- **Web**: New cache decorator and cache backends

  The cache decorator can be used to cache views, supporting both in-memory and Redis for storing the cache.

```
from faust import web

blueprint = web.Blueprint('users')
cache = blueprint.cache(timeout=300.0)

@blueprint.route('/', name='list')
class UserListView(web.View):

    @cache.view()
    async def get(self, request: web.Request) -> web.Response:
        return web.json(...)

@blueprint.route('/{user_id}/', name='detail')
class UserDetailView(web.View):

    @cache.view(timeout=10.0)
    async def get(self,
                  request: web.Request,
                  user_id: str) -> web.Response:
        return web.json(...)
```

  At this point the views are realized and can be used from Python code, but the cached `get` method handlers cannot be called yet.

To actually use the view from a web server, we need to register the blueprint to an app:

```
app = faust.App(
    'name',
    broker='kafka://',
    cache='redis://',
)
app.web.blueprints.add('/user/', 'where.is:user_blueprint')
```

After this the web server will have fully-realized views with actually cached method handlers.

The blueprint is registered with a prefix, so the URL for the `UserListView` is now `/user/`, and the URL for the `UserDetailView` is `/user/{user_id}/`.

1.10.8 Change history for Faust 1.1

This document contain change notes for bugfix releases in the Faust 1.1.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.1.3

release-date 2018-09-21 4:23 P.M PDT

release-by Ask Solem (@ask)

- **Producer**: Producing messages is now 8x to 20x faster.
- **Stream: The *`stream_publish_on_commit`* setting**

  is now disabled by default.

  Some agents produce data into topics: they forward data after processing or modify tables requiring changelog events to be sent.

  Kafka's at-least-once delivery guarantee means we will never lose a message, and we can be certain any event sent to the source topic will be processed. It also means any source event can be processed multiple times.

  If the source event is processed many times and part of the agents processing includes forwarding that event, or producing a new kind of event, then that will also happen as many times as the source event is reprocessed.

  The *`stream_publish_on_commit`* setting attempts to minimize the chances of duplicate messages being produced, by buffering up any events sent in the agent and holding on to it until the offset of the source event is committed.

  Here's an agent forwarding values to another topic:

```
@app.agent(source_topic)
async def forward(stream):
    async for value in stream:
        await destination_topic.send(value=value)
```

  If we execute this with *`stream_publish_on_commit`* enabled, then the send operation will be delayed until we have committed the offset for the source event.

  This works well when we commit often, but completely falls apart if the buffer grows too large and we have too much to do during commit.

  The commit operation works like this (in pseudo code) when *`stream_publish_on_commit`* is enabled:

```
async def commit(self):
    committable_offsets: Dict[TopicPartition, int] = ...
    # Operation A (send buffered messages related to source offsets)
    for tp, offset in committable_offsets.items():
        send_messages_buffered_up_until_offset(tp, offset)
    # Operation B (actually tell Kafka the new offsets)
    consumer.commit(committable_offsets)
```

This is not an atomic operation - the worker could crash between completing Operation A and Operation B. If there are 1000 messages to send, it could send 500 of them then crash without committing.

In this case we end up with 500 duplicate messages when the source offsets are reprocessed. Is this safer than producing one and one, and committing fast? Probably not.

That said, if you make sure the buffer never grows too large then you can take advantage of this setting to actually reduce the number of duplicate messages sent when a source topic is reprocessed.

If you want to experiment with this, tweak the *broker_commit_every* and *broker_commit_interval* settings:

```
app = faust.App('name',
                broker_commit_every=100,
                broker_commit_interval=1.0,
                stream_publish_on_commit=True)
```

The good news is that Kafka transactions are on the horizon. As soon as we have support in a Python client, we can perform this atomically, and without the overhead of buffering up messages until commit time (note from future: "exactly-once" was implemented in Faust 1.5).

1.1.2

release-date 2018-09-19 5:09 P.M PDT

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 1.17.3.
- **Agent**: Agents having concurrency=n was executing events n times.

  An unrelated change caused these additional actors to have separate channels, when they should share the same channel.

  The only tests verifying this was using mocks, so we've added a new functional test in `t/functional/agents` to be sure it won't happen again.

  This test also demonstrated a case of starvation when using concurrency: a single concurrency slot could starve others from doing work. To fix this a `sleep(0)` was added to `Stream.__aiter__`, this could improve performance in general for workers with many agents.

  Huge thanks to Zhy on the Faust slack channel for testing and identifying this issue.
- **Agent**: Less logging noise when using `concurrency`.

  This removes the additionally emitted "Starting…"/"Stopping…" logs, especially noisy with `@app.agent(concurrency=1000)`.

1.1.1

release-date 2018-09-17 4:06 P.M PDT

release-by Ask Solem (@ask)

- **Requirements**
  - Now depends on Mode 1.17.2.
- **Web: Blueprint registered to app with URL prefix would end up** having double-slash.
- **Documentation: Added *project layout suggestions*** to the application user guide.
- **Types**: annotations now passing checks on mypy 0.630.

1.1.0

release-date 2018-09-14 1:07 P.M PDT

release-by Ask Solem (@ask)

Important Notes

- **API**: Agent/Channel.send now support keyword-only arguments only

  Users often make the mistake of doing:

  ```
  channel.send(x)
  ```

  and expect that to send `x` as the value.

  But the signature is `(key, value, ...)`, so it ends up being `channel.send(key=x, value=None)`.

  Fixing this will come in two parts:

  1) Faust 1.1 (this change): Make them keyword-only arguments

     This will make it an error if the names of arguments are not specified:

     ```
     channel.send(key, value)
     ```

     Needs to be changed to:

     ```
     channel.send(key=key, value=value)
     ```

  2) **Faust 1.2: We will change the signature** to `channel.send(value, key=key, ...)`

     At this stage all existing code will have changed to using keyword-only arguments.

- **App**: The default key serializer is now `raw` (Issue #142).

  The default *value* serializer will still be `json`, but for keys it does not make as much sense to use json as the default: keys are very rarely expressed using complex structures.

  If you depend on the Faust 1.0 behavior you should override the default key serializer for the app:

  ```
  app = faust.App('myapp', ..., key_serializer='json')
  ```

  Contributed by Allison Wang (@allisonwang)

- No longer depends on click_completion

  If you want to use the shell completion command, you now have to install that dependency locally first:

```
$ ./examples/withdrawals.py completion
Usage: withdrawals.py completion [OPTIONS]

Error: Missing required dependency, but this is easy to fix.
Run `pip install click_completion` from your virtualenv
and try again!
```

  Installing click_completion:

```
$ pip install click_completion
[...]
```

News

- **Requirements**
  - Now depends on Mode 1.17.1.
  - No longer depends on click_completion
- Now works with CPython 3.6.0.
- **Models**: Record: Now supports *decimals* option to convert string decimals back to Decimal

  This can be used for any model to enable "Decimal-fields":

```
class Fundamentals(faust.Record, decimals=True):
    open: Decimal
    high: Decimal
    low: Decimal
    volume: Decimal
```

  When serialized this model will use string for decimal fields (the Javascript float type cannot be used without losing precision, it is a float after all), but when deserializing Faust will reconstruct them as Decimal objects from that string.

- **Model**: Records now support custom coercion handlers.

  Coercion converts one type into another, for example from string to `datettime`, or int/string to `Decimal`.

  In models this means conversion from the serialized form back into a corresponding Python type.

  To define a model where all `UUID` fields are serialized to string, but then converted back to `UUID` objects when deserialized, do this:

```
from uuid import UUID
import faust

class Account(faust.Record, coercions={UUID: UUID}):
    id: UUID
```

  What about non-json serializable types?

  The use of UUID in this example leaves one important detail out: json doesn't support this type so how can models serialize it?

The Faust JSON serializer adds support for UUID objects by default, but if you have a custom class you would need to add that capability by adding a `__json__` handler:

```
class MyType:

    def __init__(self, value: str):
        self.value = value

    def __json__(self):
        return self.value
```

You'd get tired writing this out for every class, so why not make an abstract model subclass:

```
from uuid import UUID
import faust

class UUIDAwareRecord(faust.Record,
                      abstract=True,
                      coercions={UUID: UUID}):
    ...

class Account(UUIDAwareRecord):
    id: UUID
```

- **App**: New `ssl_context` adds authentication support to Kafka.

  Contributed by Mika Eloranta (@melor).

- **Monitor**: New Datadog monitor (Issue #160)

  Contributed by Allison Wang (@allisonwang).

- **App: `@app.task` decorator now accepts `on_leader`**

  argument (Issue #131).

  Tasks created using the `@app.task` decorator will run once a worker is fully started.

  Similar to the `@app.timer` decorator, you can now create one-shot tasks that run on the leader worker only:

  ```
  @app.task(on_leader=True)
  async def mytask():
      print('WORKER STARTED, AND I AM THE LEADER')
  ```

  The decorated function may also accept the `app` as an argument:

  ```
  @app.task(on_leader=True)
  async def mytask(app):
      print(f'WORKER FOR APP {app} STARTED, AND I AM THE LEADER')
  ```

- **App**: New `app.producer_only` attribute.

  If set the worker will start the app without consumer/tables/agents/topics.

- **App**: `app.http_client` property is now read-write.

- **Channel**: In-memory channels were not working as expected.

  - `Channel.send(key=key, value=value)` now works as expected.
  - `app.channel()` accidentally set the `maxsize` to 1 by default, creating a deadlock.

  - `Channel.send()` now disregards the *`stream_publish_on_commit`* setting.

- **Transport**: aiokafka: Support timestamp-less messages

  Fixes error when data sent with old Kafka broker not supporting timestamps:

```
[2018-08-27 08:00:49,262: ERROR]: [^--Consumer]: Drain messages raised:
    TypeError("unsupported operand type(s) for /: 'NoneType' and 'float'",
↪)
Traceback (most recent call last):
File "faust/transport/consumer.py", line 497, in _drain_messages
    async for tp, message in ait:
File "faust/transport/drivers/aiokafka.py", line 449, in getmany
    record.timestamp / 1000.0,
TypeError: unsupported operand type(s) for /: 'NoneType' and 'float'
```

  Contributed by Mika Eloranta (@melor).

- **Distribution**: `pip install faust` no longer installs the examples directory.

  Fix contributed by Michael Seifert (@seifertm)

- **Web**: Adds exception handling to views.

  A view can now bail out early via *raise self.NotFound()* for example.

- **Web**: `@table_route` decorator now supports taking key from the URL path.

  This is now used in the `examples/word_count.py` example to add an endpoint `/count/{word}/` that routes to the correct worker with that count:

```
@app.page('/word/{word}/count/')
@table_route(table=word_counts, match_info='word')
async def get_count(web, request, word):
    return web.json({
        word: word_counts[word]
    })
```

- **Web**: Support reverse lookup from view name via `url_for`

```
web.url_for(view_name, **params)
```

- **Web**: Adds support for Flask-like "blueprints"

  Blueprint is basically just a description of a reusable app that you can add to your web application.

  Blueprints are commonly used in most Flask-like web frameworks, but Flask blueprints are not compatible with e.g. Sanic blueprints.

  The Faust blueprint is not directly compatible with any of them, but that should be fine.

  To define a blueprint:

```
from faust import web

blueprint = web.Blueprint('user')

@blueprint.route('/', name='list')
class UserListView(web.View):

    async def get(self, request: web.Request) -> web.Response:
        return self.json({'hello': 'world'})
```

(continues on next page)

(continued from previous page)

```
@blueprint.route('/{username}/', name='detail')
class UserDetailView(web.View):

    async def get(self, request: web.Request) -> web.Response:
        name = request.match_info['username']
        return self.json({'hello': name})

    async def post(self, request: web.Request) -> web.Response:
        ...

    async def delete(self, request: web.Request) -> web.Response:
        ...
```

Then to add the blueprint to a Faust app you register it:

```
blueprint.register(app, url_prefix='/users/')
```

Note: You can also create views from functions (in this case it will only support GET):

```
@blueprint.route('/', name='index')
async def hello(self, request):
    return self.text('Hello world')
```

Why?

Asyncio web frameworks are moving quickly, and we want to be able to quickly experiment with different backend drivers.

Blueprints is a tiny abstraction that fit well into the already small web abstraction that we do have.

  - Documentation and examples improvements by
    - Tom Forbes (@orf).
    - Matthew Grossman (@matthewgrossman)
    - Denis Kataev (@kataev)
    - Allison Wang (@allisonwang)
    - Huyuumi (@diplozoon)

Project

- **CI**: The following Python versions have been added to the build matrix:
  - CPython 3.7.0
  - CPython 3.6.6
  - CPython 3.6.0
- **Git**:
  - All the version tags have been cleaned up to follow the format `v1.2.3`.

- New active maintenance branches: `1.0` and `1.1`.

1.10.9 Change history for Faust 1.0

This document contain change notes for bugfix releases in the Faust 1.x series. If you're looking for changes in the latest series, please visit the latest *Changes*.

For even older releases you can visit the *History* section.

1.0.30

release-date 2018-08-15 3:17 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on Mode 1.15.1.
- **Typing: `faust.types.Message.timestamp_type` is now the correct** `int`, previously it was string by message.
- **Models**: Records can now have recursive fields.

  For example a tree structure model having a field that refers back to itself:

```
class Node(faust.Record):
    data: Any
    children: List['Node']
```

- **Models: A field of type `List[Model]` no longer raises an exception** if the value provided is `None`.
- **Models**: Adds support for `--strict-optional`-style fields.

  Previously the following would work:

```
class Order(Record):
    account: Account = None
```

  The account is considered optional from a typing point of view, but only if the mypy option `--strict-optional` is disabled.

  Now that `--strict-optional` is enabled by default in mypy, this version adds support for fields such as:

```
class Order(Record):
    account: Optional[Account] = None
    history: Optional[List[OrderStatus]]
```

- **Models: Class options such as `isodates`/`include_metadata`/etc. are** now inherited from parent class.
- **Stream**: Fixed `NameError` when pushing non-Event value into stream.

1.0.29

release-date 2018-08-10 5:00 P.M PDT

release-by Vineet Goel

- **Requirements**
  - Now depends on robinhood-aiokafka 0.4.18

    The coordination routine now ensures the program stops when receiving a `aiokafka.errors.UnknownError` from the Kafka broker. This leaves recovery up to the supervisor.
- **Table**: Fixed hanging at startup/rebalance on Python 3.7 (Issue #134).

  Workaround for `asyncio` bug seemingly introduced in Python 3.7, that left the worker hanging at startup when attempting to recover a table without any data.
- **Monitor**: More efficient updating of highwater metrics (Issue #139).
- **Partition Assignor**: The assignor now compresses the metadata being passed around to all application instances for efficiency and to avoid extreme cases where the metadata is too big.

1.0.28

release-date 2018-08-08 11:25 P.M PDT

release-by Vineet Goel

- **Monitor**: Adds consumer stats such as last read offsets, last committed offsets and log end offsets to the monitor. Also added to the StatsdMonitor.
- **aiokafka**: Changes how topics are created to make it more efficient. We now are smarter about finding kafka cluster controller instead of trial and error.
- **Documentation**: Fixed links to Slack and other minor fixes.

1.0.27

release-date 2018-07-30 04:00 P.M PDT

release-by Ask Solem

- No code changes
- Fixed links to documentation in README.rst

1.0.26

release-date 2018-07-30 08:00 A.M PDT

release-by Ask Solem

- Public release.

1.0.25

release-date 2018-07-27 12:43 P.M PDT

release-by Ask Solem

- *stream_publish_on_commit* accidentally disabled by default.

  This made the rate of producing much slower, as the default buffering settings are not optimized.

- The `App.rebalancing` flag is now reset after the tables have recovered.

1.0.24

release-date 2018-07-12 6:54 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on robinhood-aiokafka 0.4.17

    This fixed an issue where the consumer would be left hanging without a connection to Kafka.

1.0.23

release-date 2018-07-11 5:00 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on robinhood-aiokafka 0.4.16
- Now compatible with Python 3.7.
- Setting *stream_wait_empty* is now disabled by default (Issue #117).
- Documentation build now compatible with Python 3.7.
  - Fixed `ForwardRef has no attribute __origin__` error.
  - Fixed `DeprecatedInSphinx2.0` warnings.
- **Web**: Adds `app.on_webserver_init(web)` callback for ability to serve static files using `web.add_static`.
- **Web**: Adds `web.add_static(prefix, fs_path)`
- **Worker**: New `App.unassigned` attribute is now set if the worker does not have any assigned partitions.
- **CLI**: Console colors was disabled by default.

1.0.22

release-date 2018-06-27 5:35 P.M PDT

release-by Vineet Goel

- **aiokafka**: Timeout for topic creation now wraps entire topic creation. Earlier this timeout was for each individual request.
- **testing**: Added stress testing suite.

1.0.21

release-date 2018-06-27 1:43 P.M PDT

release-by Ask Solem

Warning: This changes the package name of `kafka` to `rhkafka`.

- **Requirements**
  - Now depends on robinhood-aiokafka 0.4.14
  - Now depends on Mode 1.15.0.

1.0.20

release-date 2018-06-26 2:35 P.M PDT

release-by Vineet Goel

- **Monitor**: Added `Monitor.count` to add arbitrary metrics to app monitor.
- **Statsd Monitor**: Normalize agent metrics by removing memory address to avoid spamming statsd with thousands of unique metrics per agent.

1.0.19

release-date 2018-06-25 6:40 P.M PDT

release-by Vineet Goel

- **Assignor**: Fixed crash if initial state of assignment is invalid. This was causing the following error: `ValueError('Actives and Standbys are disjoint',).` during partition assignment.

1.0.18

release-date 2018-06-21 3:53 P.M PDT

release-by Ask Solem

- **Worker**: Fixed `KeyError: TopicPartition(topic='...', partition=x)` occurring during rebalance.

1.0.17

release-date 2018-06-21 3:15 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on robinhood-aiokafka 0.4.13
- We now raise an error if the official aiokafka or kafka-python is installed.

  Faust depends on a fork of aiokafka and can not be installed with the official versions of aiokafka and kafka-python.

  If you have those in requirements, please remove them from your `virtualenv` and remove them from requirements.
- **Worker**: Fixes hanging in wait_empty.

  This should also make rebalances faster.
- **Worker**: Adds timeout on topic creation.

1.0.16

release-date 2018-06-19 3:46 P.M PDT

release-by Ask Solem

- **Worker: aiokafka create topic request default timeout now set** to 20 seconds (previously it was accidentally set to 1000 seconds).
- **Worker: Fixes crash from `AssertionError` where `table._revivers`** is an empty list.
- **Distribution**: Adds `t/misc/scripts/rebalance/killer-always-same-node.sh`.

1.0.15

release-date 2018-06-14 7:36 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on robinhood-aiokafka 0.4.12
- **Worker**: Fixed problem where worker does not recover after MacBook sleeping and waking up.
- **Worker**: Fixed crash that could lead to rebalancing loop.
- **Worker**: Removed some noisy errors that weren't really errors.

1.0.14

release-date 2018-06-13 5:58 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on robinhood-aiokafka 0.4.11
- **Worker**: aiokafka's heartbeat thread would sometimes keep the worker alive even though the worker was trying to shutdown.

  An error could have happened many hours ago causing the worker to crash and attempt a shutdown, but then the heartbeat thread kept the worker from terminating.

  Now the rebalance will check if the worker is stopped and then appropriately stop the heartbeat thread.
- **Worker**: Fixed error that caused rebalancing to hang: `"ValueError: Set of coroutines/Futures is empty."`.
- **Worker**: Fixed error "Coroutine x tried to break fence owned by y"

  This was added as an assertion to see if multiple threads would use the variable at the same time.
- **Worker**: Removed logged error "not assigned to topics" now that we automatically recover from non-existing topics.
- **Tables**: Ignore `asyncio.CancelledError` while stopping standbys.
- **Distribution**: Added scripts to help stress test rebalancing in `t/misc/scripts/rebalance`.

1.0.13

release-date 2018-06-12 2:10 P.M PDT

release-by Ask Solem

- **Worker**: The Kafka fetcher service was taking too long to shutdown on rebalance.

  If this takes longer than the session timeout, it triggers another rebalance, and if it happens repeatedly this will cause the cluster to be in a state of constant rebalancing.

  Now we use future cancellation to stop the service as fast as possible.
- **Worker**: Fetcher was accidentally started too early.

  This didn't lead to any problems that we know of, but made the start a bit slower than it needs to.
- **Worker**: Fixed race condition where partitions were paused while fetching from them.
- **Worker**: Fixed theoretical race condition hang if web server started and stopped in quick succession.
- **Statsd**: The statsd monitor prematurely initialized the event loop on module import.

  We had a fix for this, but somehow forgot to remove the "hard coded super" that was set to call: `Service.__init__(self, **kwargs)`.

  The class is not even a subclass of Service anymore, and we are lucky it manifests merely when doing something drastic, like `py.test`, recursively importing all modules in a directory.

1.0.12

release-date 2018-06-06 1:34 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on Mode 1.14.1.
- **Worker**: Producer crashing no longer causes the consumer to hang at shutdown while trying to publish attached messages.

1.0.11

release-date 2018-05-31 16:41 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on Mode 1.13.0.
  - Now depends on robinhood-aiokafka

    We have forked aiokafka to fix some issues.
- Now handles missing topics automatically, so you don't have to restart the worker the first time when topics are missing.
- Mode now registers as a library having static type annotations.

  This conforms to **PEP 561** – a new specification that defines how Python libraries register type stubs to make them available for use with static analyzers like mypy and pyre-check.
- **Typing**: Faust codebase now passes `--strict-optional`.
- **Settings**: Added new settings
  - *broker_heartbeat_interval*
  - *broker_session_timeout*
- **Aiokafka: Removes need for consumer partitions lock: this fixes** rare deadlock.
- **Worker**: Worker no longer hangs for few minutes when there is an error.

1.0.10

release-date 2018-05-15 16:02 P.M PDT

release-by Vineet Goel

- **Worker**: Stop reading changelog when no remaining messages.

1.0.9

release-date 2018-05-15 15:42 P.M PDT

release-by Vineet Goel

- **Worker**: Do not stop reading standby updates.

1.0.8

release-date 2018-05-15 11:00 A.M PDT

release-by Vineet Goel

- **Tables**
  - Fixes bug due to which we were serializing `None` values while recording a key delete to the changelog. This was causing the deleted keys to never be deleted from the changelog.
  - We were earlier not persisting offsets of messages read during changelog reading (or standby recovery). This would cause longer recovery times if recovery was ever interrupted.
- **App**: Added flight recorder for consumer group rebalances for debugging.

1.0.7

release-date 2018-05-14 4:53 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on Mode 1.12.5.
- **App**: `key_type` and `value_type` can now be set to:
  - `int`: key/value is number stored as string
  - `float`: key/value is floating point number stored as string.
  - `decimal.Decimal` key/value is decimal stored as string.
- **Agent**: Fixed support for `group_by`/`through` after change to reuse the same stream after agent crashing.
- **Agent**: Fixed `isolated_partitions=True` after change in v1.0.3.

  Initialization of the agent-by-topic index was in *1.0.3* moved to the `AgentManager.start` method, but it turns out `AgentManager` is a regular class, and not a service.

  `AgentManager` is now a service responsible for starting/stopping the agents required by the app.
- **Agent**: Include active partitions in repr when `isolated_partitions=True`.
- **Agent**: Removed extraneous 'agent crashed' exception in logs.
- **CLI**: Fixed autodiscovery of commands when using `faust -A app`.
- **Consumer**: Appropriately handle closed fetcher.
- New shortcut: *`faust.uuid()`* generates UUID4 ids as string.

1.0.6

release-date 2018-05-11 11:15 A.M PDT

release-by Vineet Goel

- **Requirements**:
  - Now depends on Aiokafka 0.4.7.
- **Table**: Delete keys when raw value in changelog set to `None`

  This was resulting in deleted keys still being present with value `None` upon recovery.
- **Transports**: Crash app on CommitFailedError thrown by aiokafka.

  App would get into a weird state upon a commit failed error thrown by the consumer thread in the aiokafka driver.

1.0.5

release-date 2018-05-08 4:09 P.M PDT

release-by Ask Solem

- **Requirements**:
  - Now depends on Mode 1.12.4.
- **Agents**: Fixed problem with hanging after agent raises exception.

  If an agent raises an exception we cannot handle it within the stream iteration, so we need to restart the agent.

  Starting from this change, even though we restart the agent, we reuse the same *`faust.Stream`* object that the crashed agent was using.

  This makes recovery more seamless and there are fewer steps involved.
- **Transports**: Fixed worker hanging issue introduced in 1.0.4.

  In version *1.0.4* we introduced a bug in the round-robin scheduling of topic partitions that manifested itself by hanging with 100% CPU usage.

  After processing all records in all topic partitions, the worker would spin loop.
- **API**: Added new base class for windows: *`faust.Window`*

  There was the typing interface *`faust.types.windows.WindowT`*, but now there is also a concrete base class that can be used in for example `Mock(autospec=Window)`.
- **Tests**: Now takes advantage of the new `AsyncMock`.

1.0.4

release-date 2018-05-08 11:45 A.M PDT

release-by Vineet Goel

- **Transports**:

  In *version-1.0.2* we implemented fair scheduling in aiokafka transport such that while processing the worker had an equal chance of processing each assigned Topic. Now we also round-robin through topic partitions within topics such that the worker has an equal chance of processing message from each assigned partition within a topic as well.

1.0.3

release-date 2018-05-07 3:45 P.M PDT

release-by Ask Solem

- **Tests**:
  - Adds 5650 lines of tests, increasing test coverage to 90%.
- **Requirements**:
  - Now depends on Mode 1.12.3.
- **Development**:
  - CI now builds coverage.
  - CI now tests multiple CPython versions:
    - CPython 3.6.0
    - CPython 3.6.1
    - CPython 3.6.2
    - CPython 3.6.3
    - CPython 3.6.4
    - CPython 3.6.5
- **Backward incompatible changes**:
  - Removed `faust.Set` unused by any internal applications.
- **Fixes**:
  - `app.agents` did not forward app to *`AgentManager`*.

    The agent manager does not use the app, but fixing this in anticipation of people writing custom agent managers.
  - ***`AgentManager`*: On partitions revoked** the agent manager now makes sure there's only one call to each agents `agent.on_partitions_revoked` callback.

    This is more of a pedantic change, but could have caused problems for advanced topic configurations.

1.0.2

release-date 2018-05-03 3:32 P.M PDT

release-by Ask Solem

- **Transports**: Implements fair scheduling in aiokafka transport.

  We now round-robin through topics when processing fetched records from Kafka. This helps us avoid starvation when some topics have many more records than others, and also takes into account that different topics may have wildly varying partition counts.

  In this version when a worker is subscribed to partitions:

```
[
    TP(topic='foo', partition=0),
    TP(topic='foo', partition=1),
    TP(topic='foo', partition=2),
    TP(topic='foo', partition=3),

    TP(topic='bar', partition=0),
    TP(topic='bar', partition=1),
    TP(topic='bar', partition=2),
    TP(topic='bar', partition=3),

    TP(topic='baz', partition=0)
]
```

  Note: `TP` is short for *topic and partition*.

  When processing messages in these partitions, the worker will round robin between the topics in such a way that each topic will have an equal chance of being processed.

- **Transports**: Fixed crash in aiokafka transport.

  The worker would attempt to commit an empty set of partitions, causing an exception to be raised. This has now been fixed.

- **Stream**: Removed unused method `Stream.tee`.

  This method was an example implementation and not used by any of our internal apps.

- **Stream: Fixed bug when something raises `StopAsyncIteration` while processing the stream.**

  The Python async iterator protocol mandates that it's illegal to raise `StopAsyncIteration` in an `__aiter__` method.

  Before this change, code such as this:

```
async for value in stream:
    value = anext(other_async_iterator)
```

  where `anext` raises `StopAsyncIteration`, Python would have the outer `__aiter__` reraise that exception as:

```
RuntimeError('__aiter__ raised StopAsyncIteration')
```

  This no longer happens as we catch the `StopAsyncIteration` exception early to ensure it does not propagate.

1.0.1

release-date 2018-05-01 9:52 A.M PDT

release-by Ask Solem

- **Stream**: Fixed issue with using `break` when iterating over stream.

  The last message in a stream would not be acked if the `break` keyword was used:

```
async for value in stream:
    if value == 3:
        break
```

- **Stream**: `.take` now acks events *after* buffer processed.

  Previously the events were erroneously acked at the time of entering the buffer.

  Note: To accomplish this we maintain a list of events to ack as soon as the buffer is processed. The operation is `O(n)` where `n` is the size of the buffer, so please keep buffer sizes small (e.g. 1000).

  A large buffer will increase the chance of consistency issues where events are processed more than once.

- **Stream**: New `noack` modifier disables acking of messages in the stream.

  Use this to disable automatic acknowledgment of events:

```
async for value in stream.noack():
    # manual acknowledgment
    await stream.ack(stream.current_event)
```

  Manual Acknowledgment

  The stream is a sequence of events, where each event has a sequence number: the "offset".

  To mark an event as processed, so that we do not process it again, the Kafka broker will keep track of the last committed offset for any topic.

  This means "acknowledgment" works quite differently from other message brokers, such as RabbitMQ where you can selectively ack some messages, but not others.

  If the messages in the topic look like this sequence:

```
1 2 3 4 5 6 7 8
```

  You can commit the offset for #5, only after processing all events before it. This means you MUST ack offsets (1, 2, 3, 4) *before* being allowed to commit 5 as the new offset.

- **Stream**: Fixed issue with `.take` not properly respecting the `within` argument.

  The new implementation of take now starts a background thread to fill the buffer. This avoids having to restart iterating over the stream, which caused issues.

1.0.0

release-date 2018-04-27 4:13 P.M PDT

release-by Ask Solem

- **Models**: Raise error if `Record.asdict()` is overridden.
- **Models**: Can now override `Record._prepare_dict` to change the payload generated.

  For example if you want your model to serialize to a dictionary, but not have any fields with `None` values, you can override `_prepare_dict` to accomplish this:

```
class Quote(faust.Record):
    ask_price: float = None
    bid_price: float = None

    def _prepare_dict(self, data):
        # Remove keys with None values from payload.
        return {k: v for k, v in data.items() if v is not None}

assert Quote(1.0, None).asdict() == {'ask_price': 1.0}
```

- **Stream**: Removed annoying `Flight Recorder` logging that was too noisy.

1.10.10 Change history for Faust 0.9

This document contain historical change notes for bugfix releases in the Faust 0.x series. To see the most recent changelog please visit *Changes*.

0.9.65

release-date 2018-04-27 2:04 P.M PDT

release-by Vineet Goel

- **Producer**: New setting to configure compression.
  - See *producer_compression_type*.
- **Documentation**: New *Advanced Producer Settings* section.

0.9.64

release-date 2018-04-26 4:48 P.M PDT

release-by Ask Solem

- **Models**: Optimization for `FieldDescriptor.__get__`.
- **Serialization**: Optimization for *`faust.utils.json`*.

0.9.63

release-date 2018-04-26 04:32 P.M PDT

release-by Vineet Goel

- **Requirements**:
  - Now depends on aiokafka 0.4.5 (Robinhood fork).
- **Models**: `Record.asdict()` and `to_representation()` were slow on complicated models, so we are now using code generation to optimize them.

  Warning: You are no longer allowed to override `Record.asdict()`.

0.9.62

release-date 2018-04-26 12:06 P.M PDT

release-by Ask Solem

- **Requirements**:
  - Now depends on Mode 1.12.2.
  - Now depends on aiokafka 0.4.4 (Robinhood fork).
- **Consumer**: Fixed `asyncio.base_futures.IllegalStateError` error in commit handler.
- **CLI**: Fixed bug when invoking worker using `faust -A`.

1.11 Authors

1.11.1 Creators

Name	Email
Ask Solem	<ask@robinhood.com>
Vineet Goel	<vineet@robinhood.com>

Note: You must not solicit for free support from email addresses on this list. Ask the community for help in the Slack channel, or ask a question on Stack Overflow.

1.11.2 Committers

Arpan Shah	<arpan@robinhood.com>
Sanyam Satia	<sanyam@robinhood.com>

Contributors become committers by stepping up to the task. They can 1) triage issues, help others on the issue tracker, code reviews, Slack or mailing lists, or 2) make modifications to documentation and code. The award for doing this in any significant capacity for one year or longer, is to be added to the list of maintainers above.

1.11.3 Contributors

Allison Wang	<allison.wang@robinhood.com>
Jamshed Vesuna	<jamshed@robinhood.com>
Jaren Glover	<jaren@robinhood.com>
Jerry Li	<jerry.li@robinhood.com>
Prithvi Narasimhan	<narasimhan.prithvi@gmail.com>
Ruby Wang	<ruby.wang@robinhood.com>
Shrey Kumar Shahi	<shrey@robinhood.com>
Mika Eloranta	<mel@aiven.io>
Omar Rayward	<orayward@yahoo.com>
Alexander Oberegger	<alexander.oberegger@smaxtec.com>
Matthew Stump	<mstump@vorstella.com>
Martin Maillard	<self@martin-maillard.com>
Mattias Karlsson	<mattias@hemmabolan.se>
Matthias Wutte	<matthias.wutte@smaxtec.com>
Thibault Serot	<thibserot@gmail.com>
Ryan Whitten	<ryan@pixability.com>
Nimi Wariboko Jr	<nimiwaribokoj@gmail.com>
Chris Seto	<chriskseto@gmail.com>
Amit Ripshtos	<amit.r@qspark.prod>
Miha Troha	<miha.troha@comcom.si>
Perk Lim	<perk@robinhood.com>
Julien Surloppe	<julien@surloppe.fr>
Bob Haddleton	<bob.haddleton@nokia.com>
Nimish Telang	<nimish@telang.net>
Cesar Pantoja	<cesarpantoj@gmail.com>

Continued on next page

Table 1 – continued from previous page

Tomasz Nguyen	<me@swistofon.pl>
Artak Papikyan	<artakp@patriot1tech.com>
Andrei Tuppitcyn	<andrew.tupitsin@gmail.com>
Vikram Patki	<vpatki@wayfair.com>
Victor Miroshnikov	<me@vmiroshnikov.com>
Tobias Rauter	<tobias.rauter@smaxtec.com>
DhruvaPatil98	<dhruva.patil@galepartners.com>
Leandro Vonwerra	<leandro.vonwerra@spoud.io>
Ignacio Peluffo	<ipeluffo@gmail.com>

1.12 Glossary

acked

acking

acknowledged Acknowledgment marks a message as fully processed. It's a signal that the program does not want to see the message again. Faust advances the offset by committing after a message is acknowledged.

agent An async function that iterates over a stream. Since streams are infinite the agent will usually not end unless the program is shut down.

codec A codec encodes/decodes data to some format or encoding. Examples of codecs include Base64 encoding, JSON serialization, pickle serialization, text encoding conversion, and more.

concurrent

concurrency A concurrent process can deal with many things at once, but not necessarily execute them in *parallel*. For example a web crawler may have to fetch thousands of web pages, and can work on them concurrently.

This is distinct from *parallelism* in that the process will switch between fetching web pages, but not actually process any of them at the same time.

consumer A process that receives messages from a broker, or a process that is actively reading from a topic/channel.

event A happening in a system, or in the case of a stream, a single record having a key/value pair, and a reference to the original message object.

idempotence

idempotent

idempotency Idempotence is a mathematical property that describes a function that can be called multiple times without changing the result. Practically it means that a function can be repeated many times without unintended effects, but not necessarily side-effect free in the pure sense (compare to *nullipotent*).

Further reading: https://en.wikipedia.org/wiki/Idempotent

message The unit of data published or received from the message *transport*. A message has a key and a value.

nullipotent

nullipotence

nullipotency describes a function that'll have the same effect, and give the same result, even if called zero or multiple times (side-effect free). A stronger version of *idempotent*.

parallel

parallelism A parallel process can execute many things at the same time, which will usually require running on multiple CPU cores.

In contrast the term *concurrency* refers to something that is seemingly parallel, but does not actually execute at the same time.

publisher A process sending messages, or a process publishing data to a topic.

reentrant

reentrancy describes a function that can be interrupted in the middle of execution (e.g., by hardware interrupt or signal), and then safely called again later. Reentrancy isn't the same as *idempotence* as the return value doesn't have to be the same given the same inputs, and a reentrant function may have side effects as long as it can be interrupted; An idempotent function is always reentrant, but the reverse may not be true.

sensor A sensor records information about events happening in a running Faust application.

serializer A serializer is a *codec*, responsible for serializing keys and values in messages sent over the network.

task A task is the unit of *concurrency* in an `asyncio` program.

thread safe A function or process that is thread safe means multiple POSIX threads can execute it in parallel without race conditions or deadlock situations.

topic Consumers subscribe to topics of interest, and producers send messages to consumers via the topic.

transport A communication mechanism used to send and receive messages, for example Kafka.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W