MicroAgent Documentation

Release 1.7.3

Dmitrii Vlasov

Mar 12, 2024

CORE:

The goal of this project is to facilitate the creation of **microservices** interacting via a **signal bus** and/or **queue broker**.

The philosophy of this project is to present a microservice as a software agent that directly interacts only with queues and the event bus, and not with other microservices.

Tool is intended for developing:

- distributed apps with **event-driven** architecture
- distributed apps with **data-driven** architecture
- multi-processors apps

Tool provide features:

- running a **periodical tasks** (interval or as CRON)
- specification of signals (events), their sending and receiving via the bus (*redis*)
- description of queues, sending and receiving messages via the queue broker (amqp, kafka, redis)
- limited **RPC** via signal bus
- launching sub-services (in the same process)
- launching a group of microagents (each in a separate process)
- mocks for bus and broker

```python
class Agent(MicroAgent):

    @on('pre_start')
    async def setup(self):
        pass  # init connections to DB, REDIS, SMTP and other services

    @periodic(period=5)
    async def refresh_cache(self):
        pass  # do something periodicly

    @cron('0 */4 * * *')
    async def send_report(self):
        pass  # do something by schedule

    # subscribe to signals (events)
    @receiver(signals.user_updated, signals.user_deleted)
    async def send_notification(self, **kwargs):
        # send signal (event) to bus
        await self.bus.check_something.send(sender='agent', **kwargs)

    # message consumer from queue
    @consumer(queues.mailer)
    async def send_emails(self, **kwargs):
        # send message to queue
        await self.broker.statistic_collector.send(kwargs)


async def main():
    bus = RedisSignalBus('redis://localhost/7')
    broker = RedisBroker('redis://localhost/7')
```

(continues on next page)

(continued from previous page)

```
    # usage bus and broker separate from agent
    await bus.started.send('user_agent')
    await broker.mailer(data)

    agent = Agent(bus=bus, broker=broker)
    await agent.start()
```

CHAPTER

ONE

MICROAGENT

MicroAgent is a **container** for **signal receivers**, **message consumers** and **periodic tasks**. The MicroAgent links the abstract declarations of receivers and consumers with the provided bus and broker. The MicroAgent initiates periodic tasks and starts the servers.

The microagent can be launched using the supplied launcher. Or it can be used as a stand-alone entity running in python-shell or a custom script. It may be useful for testing, exploring, debugging and launching in specific purposes.

Agent declaration.

```
from microagent import MicroAgent, receiver, consumer, periodic, cron, on

class Agent(MicroAgent):

    @on('pre_start')
    async def setup(self):
        pass

    @periodic(period=5)
    async def periodic_handler(self):
        pass

    @receiver(signals.send_mail)
    async def send_mail_handler(self, **kwargs):
        pass

    @consumer(queues.mailer)
    async def mail_handler(self, **kwargs):
        pass
```

Agent initiation.

```
import logging
from microagent.tools.redis import RedisSignalBus, RedisBroker

# Initialize bus, broker and logger
bus = RedisSignalBus('redis://localhost/7')
broker = RedisBroker('redis://localhost/7')
log = logging.getLogger('my_log')
settings = {'secret': 'my_secret'}

# Initialize MicroAgent, all arguments optional
agent = Agent(bus=bus, broker=broker, log=logger, settings=settings)
```

Manual launching.

```
await user_agent.start()  # freezed here if sub-servers running

while True:
    await asyncio.sleep(60)
```

Using MicroAgent resources.

```
class Agent(MicroAgent):

    async def setup(self):
        self.log.info('Setup called!')  # write log
        await self.bus.my_sugnal.send(sender='agent', param=1)  # use bus
        await self.broker.my_queue.send({'text': 'Hello world!'})  # use broker
        secret = self.settings['secret']  # user settings
        print(self.info())  # serializable dict of agent structure
```

Decorators details.

microagent.**consumer**(*queue:* Queue, *timeout:* *int* = *60*, *dto_class:* *type* | *None* = *None*, *dto_name:* *str* | *None* = *None*, ***options:* *Any*) → Callable[[Callable[[...], Awaitable[None]]], Callable[[...], Awaitable[None]]]

Binding for consuming messages from queue.

Only **one** handler can be bound to **one** queue.

Parameters

- **queue** – Queue - source of data
- **timeout** – Calling timeout in seconds
- **dto_class** – DTO-class, wrapper for consuming data
- **dto_name** – DTO name in consumer method kwargs

```
@consumer(queue_1)
async def handler_1(self, **kwargs):
    log.info('Called handler 1 %s', kwargs)

@consumer(queue_2, timeout=30)
async def handle_2(self, **kwargs):
    log.info('Called handler 2 %s', kwargs)

@consumer(queue_3, dto_class=MyDTO)
async def handle_3(self, dto: MyDTO, **kwargs):
    log.info('Called handler 3 %s', dto)  # dto = MyDTO(**kwargs)

@consumer(queue_4, timeout=30, dto_class=MyDTO, dto_name='obj')
async def handle_4(self, obj: MyDTO, **kwargs):
    log.info('Called handler 4 %s', obj)  # obj = MyDTO(**kwargs)
```

microagent.**cron**(*spec:* *str*, *timeout:* *int* | *float* = *1*) → Callable[[Callable[[Any], Awaitable[None]]], Callable[[Any], Awaitable[None]]]

Run decorated function by schedule (cron)

Parameters

- **spec** – Specified running scheduling in cron format
- **timeout** – Function timeout in seconds

```
@periodic('0 */4 * * *')
async def handler_1(self):
    log.info('Called handler 1')

@periodic('*/15 * * * *', timeout=10)
async def handler_2(self):
    log.info('Called handler 2')
```

microagent.**on**(*label: str*) → Callable[[Callable[[Any], Awaitable[None]]], Callable[[Any], Awaitable[None]]]

Hooks for internal events *(pre_start, post_start, pre_stop)* or running forever servers *(server)*.

Server-function will be call as run-forever asyncio task.

Parameters

label – Hook type label string *(pre_start, post_start, pre_stop, server)*

```
@on('pre_start')
async def handler_1(self):
    log.info('Called handler 1')

@on('post_start')
async def handler_2(self):
    log.info('Called handler 2')

@on('pre_stop')
async def handler_3(self):
    log.info('Called handler 3')

@on('server')
async def run_server(self):
    await Server().start()  # run forever
    raise ServerInterrupt('Exit')  # graceful exit
```

microagent.**periodic**(*period: int | float*, *timeout: int | float = 1*, *start_after: int | float = 0*) → Callable[[Callable[[Any], Awaitable[None]]], Callable[[Any], Awaitable[None]]]

Run decorated handler periodically.

Parameters

- **period** – Period of running functions in seconds
- **timeout** – Function timeout in seconds
- **start_after** – Delay for running loop in seconds

```
@periodic(period=5)
async def handler_1(self):
    log.info('Called handler 1')

@periodic(5, timeout=4)
async def handler_2(self):
    log.info('Called handler 2')
```

(continues on next page)

(continued from previous page)

```
@periodic(period=5, start_after=10)
async def handler_3(self):
    log.info('Called handler 3')
```

microagent.**receiver**(**signals:* Signal, *timeout:* *int* = *60*) → Callable[[Callable[[...], Awaitable[None | int | str]]], Callable[[...], Awaitable[None | int | str]]]

Binding for signals receiving.

Handler can receive **many** signals, and **many** handlers can receiver same signal.

Parameters

- **signals** – List of receiving signals
- **timeout** – Calling timeout in seconds

```
@receiver(signal_1, signal_2)
async def handler_1(self, **kwargs):
    log.info('Called handler 1 %s', kwargs)

@receiver(signal_1)
async def handle_2(self, **kwargs):
    log.info('Called handler 2 %s', kwargs)

@receiver(signal_2, timeout=30)
async def handle_3(self, **kwargs):
    log.info('Called handler 3 %s', kwargs)
```

class microagent.**MicroAgent**(*bus: ~microagent.bus.AbstractSignalBus | None = None*, *broker: ~microagent.broker.AbstractQueueBroker | None = None*, *log: ~logging.Logger = <factory>*, *settings: dict = <factory>*)

MicroAgent is a **container** for **signal receivers**, **message consumers** and **periodic tasks**.

The magic of declarations binding to an agent-object is implemented in *__new__*. Attaching the bus, broker and logger is implemented in *__init__*. Subscribing and initiating periodic tasks and servers is implemented in *start* method.

To create specialized MicroAgent classes, you can override *__init__*, which is safe for the constructor logic. But it is usually sufficient to use *@on('pre_start')* decorator for sync or async methods for initializing resources and etc.

Parameters

- **bus** – signal bus, object of subclass `AbstractSignalBus`, required for receive or send the signals
- **broker** – queue broker, object of subclass `AbstractQueueBroker`, required for consume or send the messages
- **log** – prepared `logging.Logger`, or use default logger if not provided
- **settings** – dict of user settings storing in object

log

Prepared python logger:

```
self.log.info('Hellow world')
```

bus

Signal bus, provided on initializing:

```
await self.bus.send_mail.send('agent', user_id=1)
```

broker

Queue broker, provided on initializing:

```
await self.broker.mailer.send({'text': 'Hello world'})
```

settings

Dict, user settings, provided on initializing, or empty.

async start(*enable_periodic_tasks: bool = True*, *enable_receiving_signals: bool = True*, *enable_consuming_messages: bool = True*, *enable_servers_running: bool = True*) → None

Starting MicroAgent to receive signals, consume messages and initiate periodic running.

Parameters

- **enable_periodic_tasks** – default enabled
- **enable_consuming_messages** – default enabled
- **enable_receiving_signals** – default enabled
- **enable_servers_running** – default enabled

async bind_receivers(*receivers: Iterable[Receiver]*) → None

Bind signal receivers to bus subscribers

async bind_consumers(*consumers: Iterable[Consumer]*) → None

Bind message consumers to queues

info() → dict

Information about MicroAgent in json-serializable dict

CHAPTER

TWO

PERIODIC FUNCTIONS

The MicroAgent method can be run periodically after a certain period of time or on a schedule (cron).

Periodic calls are implemented with *asyncio.call_later* chains. Before each method call, the next call is initiated. Each call is independent, and previous calls do not affect subsequent calls. Exceptions are written to the logger in the associated Microagent.

```
class Agent(MicroAgent):

    @periodic(period=3, timeout=10, start_after=2)  # in seconds
    async def periodic_handler(self):
        pass  # code here

    @cron('*/10 * * * *', timeout=10)  # in seconds
    async def cron_handler(self):
        pass  # code here
```

microagent.**cron**(*spec: str*, *timeout: int | float = 1*) → Callable[[Callable[[Any], Awaitable[None]]], Callable[[Any], Awaitable[None]]]

Run decorated function by schedule (cron)

Parameters

- **spec** – Specified running scheduling in cron format
- **timeout** – Function timeout in seconds

```
@periodic('0 */4 * * *')
async def handler_1(self):
    log.info('Called handler 1')

@periodic('*/15 * * * *', timeout=10)
async def handler_2(self):
    log.info('Called handler 2')
```

microagent.**periodic**(*period: int | float*, *timeout: int | float = 1*, *start_after: int | float = 0*) → Callable[[Callable[[Any], Awaitable[None]]], Callable[[Any], Awaitable[None]]]

Run decorated handler periodically.

Parameters

- **period** – Period of running functions in seconds
- **timeout** – Function timeout in seconds
- **start_after** – Delay for running loop in seconds

```
@periodic(period=5)
async def handler_1(self):
    log.info('Called handler 1')

@periodic(5, timeout=4)
async def handler_2(self):
    log.info('Called handler 2')

@periodic(period=5, start_after=10)
async def handler_3(self):
    log.info('Called handler 3')
```

class microagent.timer.**PeriodicTask**(*agent: 'MicroAgent'*, *handler: collections.abc.Callable[[Any], collections.abc.Awaitable[None]]*, *period: float*, *timeout: float*, *start_after: float*)

class microagent.timer.**CRONTask**(*agent: 'MicroAgent'*, *handler: collections.abc.Callable[[Any], collections.abc.Awaitable[None]]*, *cron: microagent.timer.CRON*, *timeout: float*)

property start_after: float

start_after property of **CRONTask** object is a next value of generator behind facade. Be carefully with manual manipulation with it.

property period: float

period property of **CRONTask** object is a next value of generator behind facade. Be carefully with manual manipulation with it.

CHAPTER

THREE

SIGNAL BUS

Event-driven architecture is based on objects exchanging a non-directed messages - events. Here we assume that events (signals) that are not stored anywhere, everyone can receive and send them, like a radio-transfer.

Here, the intermediary that manages messages routing is called the Signal Bus and implements the publish / subscribe pattern. A signal is a message with a strict fixed structure. The Bus contains many channels that are different for each type of signal. We can send a signal from many sources and listen it with many receivers.

Implementations:

- *redis*

Using SignalBus separately (sending only)

```
from microagent import load_signals
from microagent.tools.redis import RedisSignalBus

signals = load_signals('file://signals.json')

bus = RedisSignalBus('redis://localhost/7')
await bus.user_created.send('user_agent', user_id=1)
```

Using with MicroAgent

```
from microagent import MicroAgent, load_signals
from microagent.tools.redis import RedisSignalBus

signals = load_signals('file://signals.json')

class UserAgent(MicroAgent):
    @receiver(signals.user_created)
    async def example(self, user_id, **kwargs):
        await self.bus.user_created.send('some_signal', user_id=1)

bus = RedisSignalBus('redis://localhost/7')
user_agent = UserAgent(bus=bus)
await user_agent.start()
```

microagent.**load_signals**(*source: str*) → NamedTuple

Load Signal-entities from file or by web.

```
from microagent import load_signals
```

(continues on next page)

(continued from previous page)

```
signals_from_file = load_signals('file://signals.json')
signals_from_web = load_signals('http://example.com/signals.json')
```

Signals declarations (signals.json).

```
{
    "signals": [
        {"name": "started", "providing_args": []},
        {"name": "user_created", "providing_args": ["user_id"]},
        {"name": "typed_signal", "providing_args": {
            "uuid": "string",
            "code": ["number", "null"],
            "flag": "boolean",
            "ids": "array"
        }}
    ]
}
```

microagent.**receiver**(**signals:* Signal, *timeout:* *int* = *60*) → Callable[[Callable[[...], Awaitable[None | int | str]]], Callable[[...], Awaitable[None | int | str]]]

Binding for signals receiving.

Handler can receive **many** signals, and **many** handlers can receiver same signal.

Parameters

- **signals** – List of receiving signals
- **timeout** – Calling timeout in seconds

```
@receiver(signal_1, signal_2)
async def handler_1(self, **kwargs):
    log.info('Called handler 1 %s', kwargs)

@receiver(signal_1)
async def handle_2(self, **kwargs):
    log.info('Called handler 2 %s', kwargs)

@receiver(signal_2, timeout=30)
async def handle_3(self, **kwargs):
    log.info('Called handler 3 %s', kwargs)
```

class microagent.bus.**AbstractSignalBus**(*dsn: str*, *uid: str* = <*factory*>, *prefix: str* = *'PUBSUB'*, *log: ~logging.Logger* = <*Logger microagent.bus (WARNING)*>, *receivers: dict[str, list[~microagent.signal.Receiver]]* = <*factory*>, *_responses: dict[str, ~microagent.utils.IterQueue]* = <*factory*>)

Signal bus is an abstract interface with two basic methods - send and bind.

send-method allows to publish some signal in the channel for subscribers.

bind-method allows to subscribe to the channel(s) for receive the signal(s).

call-method allows to use RPC based on *send* and *bind*.

All registered Signals are available in the bus object as attributes with the names specified in the declaration.

```
Signal(name='user_created', providing_args=['user_id'])

bus = RedisSignalBus('redis://localhost/7')
await bus.user_created.send('user_agent', user_id=1)
```

dsn

Bus has only one required parameter - dsn-string (data source name), which provide details for establishing a connection with the mediator-service.

prefix

Channel prefix, string, one for project. It is allowing use same redis for different projects.

log

Provided or defaul logger

uid

UUID, id of bus instance (for debugging)

receivers

Dict of all binded receivers

abstract async send(*channel: str*, *message: str*) → None

Send raw message to channel. Available optional type checking for input data.

Parameters

- **channel** – string, channel name
- **message** – string, serialized object

abstract async bind(*signal: str*) → None

Subscribe to channel.

Parameters
signal – string, signal name for subscribe

async bind_receiver(*receiver:* Receiver) → None

Bind bounded to agent receiver to current bus.

call(*channel: str*, *message: str*, *timeout: int*) → AsyncIterator[IterQueue]

RPC over pub/sub. Pair of signals - sending and responsing. Response-signal is an internal construction enabled by default. When we call *call* we send a ordinary declared by user signal with a unique id and awaiting a response with same id. The response can contain a string value or an integer that is returned by the signal receiver.

Signal-attached method *call* will catch only first value. To process multiple responses, you can use async context *waiter*, which will return an async generator of response data. You can break it or return value when it needed. *waiter*-method has the *timeout* argument set to 60 by default.

Answer: `{"<Class>.<method>": <value>}`

Available optional type checking for input data.

```
class CommentAgent(MicroAgent):
    @receiver(signals.rpc_comments_count)
    async def example_rpc_handler(self, user_id, **kwargs):
        return 1
```

(continues on next page)

(continued from previous page)

```
response = await bus.rpc_comments_count.call('user_agent', user_id=1)
value = response['CommentAgent.example_rpc_handler']

async with bus.rpc_comments_count.waiter('user_agent', user_id=1) as queue:
    async for x in queue:
        logging.info('Get response %s', x)
        break
```

receiver(*channel: str*, *message: str*) → None

Handler of raw incoming messages. Available optional type checking for input data.

class microagent.bus.**Signal**(*name: str*, *providing_args: list[str]*, *type_map: dict[str, tuple[type, ...]] | None = None*)

Dataclass (declaration) for a signal entity with a unique name. Each instance registered at creation. Usually, you don't need to work directly with the Signal-class.

name

String, signal name, project-wide unique, *[a-z_]+*

providing_args

All available and required parameters of message, can be simple list of argument names, or dictionary with declared types for each argument. If types declared, will be enabled soft type checking (warning log) for input data in runtime. Type checking works in *bus.send*, *bus.call* and on receiving signals. Supported only json-types: string, number, boolean, array, object, null.

Declaration with config-file (signals.json).

```
{
    "signals": [
        {"name": "started", "providing_args": []},
        {"name": "user_created", "providing_args": ["user_id"]},
        {"name": "typed_signal", "providing_args": {
            "uuid": "string",
            "code": ["number", "null"],
            "flag": "boolean",
            "ids": "array"
        }}
    ]
}
```

Manual declaration (not recommended)

```
some_signal = Signal(
    name='some_signal',
    providing_args=['some_arg']
)
```

classmethod get(*name: str*) → *Signal*

Get the signal instance by name

classmethod get_all() → dict[str, *Signal*]

All registered signals

make_channel_name(*channel_prefix: str*, *sender: str = '*'*) → str

Construct a channel name by the signal description

Parameters

- **channel_prefix** – prefix, often project name
- **sender** – name of signal sender

serialize(*data: dict*) → str

Data serializing method

Parameters

data – dict of transfered data

deserialize(*data: str*) → dict

Data deserializing method

Parameters

data – serialized transfered data

Internals stuff for signal bus binding

class microagent.bus.**BoundSignal**(*bus:* microagent.bus.AbstractSignalBus, *signal:* microagent.signal.Signal)

waiter(*sender: str*, *timeout: int = 60*, ***kwargs: Any*) → AbstractAsyncContextManager

async with bus.iter(sender='name', a=1, timeout=10) as queue:

async for x in queue:

logging.info('Get response %s', x) break

class microagent.bus.**Receiver**(*agent: 'MicroAgent'*, *handler: collections.abc.Callable[..., collections.abc.Awaitable[None | int | str]]*, *signal:* microagent.signal.Signal, *timeout: float*)

Exceptions

class microagent.signal.**SignalException**

Base signal exception

class microagent.signal.**SignalNotFound**

class microagent.signal.**SerializingError**

CHAPTER

FOUR

QUEUE BROKER

The data-driven architecture is based on unidirectional message flows between agents. Here we assume that messages are exchanged through an intermediary, not directly.

Here, an intermediary called Queue Broker implements the producer / consumer pattern. The broker performs the functions of guaranteed and consistent transmission of messages from the product to the consumer, many to one (or according to the broker's own logic). The message has a free structure, fully defined in the user area.

Implementations:

- *redis*
- *aioamqp*
- *kafka*

Using QueueBroker separately (sending only)

```
from microagent import load_queues
from microagent.tools.redis import RedisBroker

queues = load_queues('file://queues.json')

broker = RedisBroker('redis://localhost/7')
await broker.user_created.send({'user_id': 1})
```

Using with MicroAgent

```
from microagent import MicroAgent, load_queues
from microagent.tools.redis import RedisSignalBus

queues = load_queues('file://queues.json')

class EmailAgent(MicroAgent):
    @consumer(queues.mailer)
    async def example_read_queue(self, **kwargs):
        await self.broker.email_sended.send({'user_id': 1})

broker = RedisBroker('redis://localhost/7')
email_agent = EmailAgent(broker=broker)
await email_agent.start()
```

microagent.**load_queues**(*source: str*) → NamedTuple

Load Queue-entities from file or by web.

```
from microagent import load_queues

signals_from_file = load_signals('file://queues.json')
signals_from_web = load_signals('http://example.com/queues.json')
```

Queues declarations (queues.json).

```
{
    "queues": [
        {"name": "mailer"},
        {"name": "pusher"},
    ]
}
```

microagent.**consumer**(*queue: Queue*, *timeout: int = 60*, *dto_class: type | None = None*, *dto_name: str | None = None*, ***options: Any*) → Callable[[Callable[[...], Awaitable[None]]], Callable[[...], Awaitable[None]]]

Binding for consuming messages from queue.

Only **one** handler can be bound to **one** queue.

Parameters

- **queue** – Queue - source of data
- **timeout** – Calling timeout in seconds
- **dto_class** – DTO-class, wrapper for consuming data
- **dto_name** – DTO name in consumer method kwargs

```
@consumer(queue_1)
async def handler_1(self, **kwargs):
    log.info('Called handler 1 %s', kwargs)

@consumer(queue_2, timeout=30)
async def handle_2(self, **kwargs):
    log.info('Called handler 2 %s', kwargs)

@consumer(queue_3, dto_class=MyDTO)
async def handle_3(self, dto: MyDTO, **kwargs):
    log.info('Called handler 3 %s', dto)  # dto = MyDTO(**kwargs)

@consumer(queue_4, timeout=30, dto_class=MyDTO, dto_name='obj')
async def handle_4(self, obj: MyDTO, **kwargs):
    log.info('Called handler 4 %s', obj)  # obj = MyDTO(**kwargs)
```

class microagent.broker.**AbstractQueueBroker**(*dsn: str*, *uid: str = <factory>*, *log: ~logging.Logger = <Logger microagent.broker (WARNING)>*, *_bindings: dict[str, ~microagent.queue.Consumer] = <factory>*)

Broker is an abstract interface with two basic methods - send and bind.

send-method allows to write a message to the queue.

bind-method allows to connect to queue for reading.

All registered Queue are available in the broker object as attributes with the names specified in the declaration.

```
Queue(name='user_created')

broker = RedisBroker('redis://localhost/7')
await broker.user_created.send({'user_id': 1})
```

dsn

Broker has only one required parameter - dsn-string (data source name), which provide details for establishing connection to mediator-service.

log

Provided or defaul logger

uid

UUID, id of broker instance (for debugging)

_bindings

Dict of all bound consumers

abstract async send(*name: str*, *message: str*, ***kwargs: Any*) → None

Write a raw message to queue.

Parameters

- **name** – string, queue name
- **message** – string, serialized object
- ****kwargs** – specific parameters for each broker implementation

abstract async bind(*name: str*) → None

Start reading queue.

Parameters
name – string, queue name

async bind_consumer(*consumer:* Consumer) → None

Bind bounded to agent consumer to current broker.

abstract async queue_length(*name: str*, ***options: Any*) → int

Get the current queue length.

Parameters

- **name** – string, queue name
- ****options** – specific parameters for each broker implementation

class microagent.queue.**Queue**(*name: str*)

Dataclass (declaration) for a queue entity with a unique name. Each instance registered at creation. Usually, you don't need to work directly with the Queue-class.

name

String, queue name, project-wide unique, *[a-z_]+*

Declaration with config-file (queues.json)

```
{
    "queues": [
        {"name": "mailer"},
```

(continues on next page)

(continued from previous page)

```
        {"name": "pusher"},
    ]
}
```

Manual declaration (not recommended)

```
some_queue = Queue(
    name='some_queue'
)
```

classmethod get(*name: str*) → *Queue*

Get the queue instance by name

classmethod get_all() → dict[str, *Queue*]

All registered queues

serialize(*data: dict*) → str

Data serializing method

Parameters
data – dict of transfered data

deserialize(*data: str | bytes*) → dict

Data deserializing method

Parameters
data – serialized transfered data

Internals stuff for queues broker binding

class microagent.broker.**BoundQueue**(*broker: 'AbstractQueueBroker'*, *queue:* microagent.queue.Queue)

class microagent.broker.**Consumer**(*agent: 'MicroAgent'*, *handler: collections.abc.Callable[..., collections.abc.Awaitable[None]]*, *queue:* microagent.queue.Queue, *timeout: float*, *options: dict*, *dto_class: type | None = None*, *dto_name: str | None = None*)

Exceptions

class microagent.queue.**QueueException**

Base queue exception

class microagent.queue.**QueueNotFound**

class microagent.queue.**SerializingError**

CHAPTER

FIVE

INTERNAL HOOKS

In practice, it is useful to be able to perform some actions before the microagent starts working or after it stops. For this aim there are internal hooks that allow you to run methods on pre_start, post_start, and pre_stop.

pre_start - is called before the microagent is ready to accept events and consume messages. This ensures that handlers will be called when already connections established with other services - databases, mail, logs; initialized caches, objects, and so on.

post_start - called when the microagent has already started accepting events and messages. It is can be useful for sending notifications to monitoring service and etc.

pre_stop - called when the microagent go shutdown. It can be useful for sending notifications to the monitoring service, and so on.

server - "run forever" handler. If it crashes with exception microagent will be stopped. If you are using a launcher from the library and server run forever, it is important correctly to stop the servers with ServerInterrupt exception.

In addition, there is a special mechanism for running nested services. Methods marked with the server decorator will be started in "run forever" mode. It's allow provide endpoints for microagent, such as http, websocket, smtp or other.

`microagent.on`(*label: str*) → Callable[[Callable[[Any], Awaitable[None]]], Callable[[Any], Awaitable[None]]]

Hooks for internal events *(pre_start, post_start, pre_stop)* or running forever servers *(server)*.

Server-function will be call as run-forever asyncio task.

Parameters

label – Hook type label string *(pre_start, post_start, pre_stop, server)*

```
@on('pre_start')
async def handler_1(self):
    log.info('Called handler 1')

@on('post_start')
async def handler_2(self):
    log.info('Called handler 2')

@on('pre_stop')
async def handler_3(self):
    log.info('Called handler 3')

@on('server')
async def run_server(self):
    await Server().start()  # run forever
    raise ServerInterrupt('Exit')  # graceful exit
```

CHAPTER

SIX

LAUNCHER AND CONFIGURATION

Configuration and launch MicroAgents with shipped launcher. Configuration file is a python-file with 3 dictionaries: AGENT, BUS and BROKER, where specified all settings. Launcher can run microagents from one or several files.

```
$ marun myproject.app1 myproject.app2
```

Each microagent is launched in a separate os process, and the launcher works as a supervisor. All agents started by a single command are called a deployment group and start/stop at the same time. If one of the agents stops, the launcher stops the entire group.

```
import sys
import logging

logging.basicConfig(format=(
    '%(levelname)-8s [pid#%(process)d] %(asctime)s %(name)s '
    '%(filename)s:%(lineno)d %(message)s'
), stream=sys.stdout, level=logging.DEBUG)


BUS = {
    'redis': {
        'backend': 'microagent.tools.redis.RedisSignalBus',
        'dsn': 'redis://localhost/7',
        'prefix': 'PREF',
    },
}

BROKER = {
    'redis': {
        'backend': 'microagent.tools.redis.RedisBroker',
        'dsn': 'redis://localhost/7',
    },
}


AGENT = {
    'user_agent': {
        'backend': 'examples.user_agent.UserAgent',
        'bus': 'redis',
        'broker': 'redis',
    },
    'comment_agent': {
```

(continues on next page)

(continued from previous page)

```
        'backend': 'examples.comment_agent.CommentAgent',
        'bus': 'redis',
        'broker': 'redis',
    },
    'email_agent': {
        'backend': 'examples.email_agent.EmailAgent',
        'broker': 'redis',
    },
}
```

class microagent.launcher.**ServerInterrupt**

Graceful server interruption

microagent.launcher.**load_configuration**(*config_path: str*) → Iterator[tuple[str, tuple[str, dict[str, Any]]]]

Load configuration from module and prepare it for initializing agents. Returns list of unfolded configs for each agent.

microagent.launcher.**init_agent**(*backend: str*, *cfg: dict[str, Any]*) → *MicroAgent*

Import and load all using backends from config, initialize it and returns not started MicroAgent instance

class microagent.launcher.**AgentsManager**(*cfg: list[tuple[str, tuple[str, dict[str, Any]]]]*)

AgentsManager is a supervisor for launching and control group of microagents. When we run AgentsManager, it fork daemon process, strat microagent in the it, and wait when it finished or failed, then send SIGTERM for all other working processes.

class microagent.launcher.**GroupInterrupt**

CHAPTER

SEVEN

REDIS

Signal Bus and *Queue Broker* based on redis.

class microagent.tools.redis.**RedisSignalBus**(*dsn: str*, *uid: str = <factory>*, *prefix: str = 'PUBSUB'*, *log: ~logging.Logger = <Logger microagent.bus (WARNING)>*, *receivers: dict[str, list[~microagent.signal.Receiver]] = <factory>*, *_responses: dict[str, ~microagent.utils.IterQueue] = <factory>*, *_pubsub_lock: ~asyncio.locks.Lock = <factory>*)

Bus is based on redis publish and subscribe features. Channel name is forming by rule `{prefix}:{signal_name}:{sender_name}#{message_id}`

Example:

```
from microagent.tools.redis import RedisSignalBus

bus = RedisSignalBus('redis://localhost/7', prefix='MYAPP', log=custom_logger)

await bus.user_created.send('user_agent', user_id=1)
```

async send(*channel: str*, *message: str*) → None

Send raw message to channel. Available optional type checking for input data.

Parameters

- **channel** – string, channel name
- **message** – string, serialized object

async bind(*channel: str*) → None

Subscribe to channel.

Parameters

signal – string, signal name for subscribe

class microagent.tools.redis.**RedisBroker**(*dsn: str*, *uid: str = <factory>*, *log: ~logging.Logger = <Logger microagent.broker (WARNING)>*, *_bindings: dict = <factory>*, *WAIT_TIME: int = 15*, *BIND_TIME: float = 1*, *ROLLBACK_ATTEMPTS: int = 3*, *_rollbacks: dict = <factory>*)

Broker is based on Redis lists and RPUSH and BLPOP commands. Queue name using as a key. If hanling faild, message will be returned to queue 3 times (by default) and then droped.

Example:

```
from microagent.tools.redis import RedisBroker

broker = RedisBroker('redis://localhost/7', log=custom_logger)

await broker.user_created.send({'user_id': 1})
```

ROLLBACK_ATTEMPTS

Number attempts for handling of message before it will be droped (by default: 3)

WAIT_TIME

BLPOP option (by default: 15)

async send(*name: str*, *message: str*, ***kwargs: Any*) → None

Write a raw message to queue.

Parameters

- **name** – string, queue name
- **message** – string, serialized object
- ****kwargs** – specific parameters for each broker implementation

async queue_length(*name: str*, ***options: Any*) → int

Get the current queue length.

Parameters

- **name** – string, queue name
- ****options** – specific parameters for each broker implementation

async bind(*name: str*) → None

Start reading queue.

Parameters

name – string, queue name

CHAPTER

EIGHT

AMQP (RABBITMQ)

Queue Broker based on aioamqp.

class microagent.tools.amqp.**AMQPBroker**(*dsn: str*, *uid: str = <factory>*, *log: ~logging.Logger = <Logger microagent.broker (WARNING)>*, *_bindings: dict[str, ~microagent.queue.Consumer] = <factory>*, *sending_channel: ~aiormq.abc.AbstractChannel | None = None*)

The broker is based on the basic_consume method of the AMQP and sends a acknowledgement automatically if the handler is completed without errors. The consumer takes an exclusive channel. Sending an reuse the channels.

Parameters

- **dsn** – string, data source name for connection amqp://guest@localhost:5672/
- **log** – logging.Logger (optional)

```
from microagent.tools.amqp import AMQPBroker

broker = AMQPBroker('amqp://guest:guest@localhost:5672/')

await broker.user_created.send({'user_id': 1})
```

@*consumer*-decorator for this broker has an additional option - *autoack*, which enables / disables sending automatic acknowledgements.

```
class EmailAgent(MicroAgent):
    @consumer(queues.mailer, autoack=False)
    async def example_read_queue(self, amqp, **data):
        await amqp.channel.basic_client_ack(delivery_tag=amqp.delivery_tag)
```

Handler will takes one required positional argument - pamqp.DeliveredMessage. Consumer will be reconnect and subscribe to queue on disconnect. It make 3 attempts of reconnect after 1, 4, 9 seconds. if the queue does not exist, it will be declared with the default parameters when binding.

async bind(*name: str*) → None

Start reading queue.

Parameters

name – string, queue name

async declare_queue(*name: str*, ***options: Any*) → None

Declare queue with queue_declare method.

Parameters

- **name** – string, queue name
- ****options** – other queue_declare options

async get_channel() → AbstractChannel

Takes a channel from the pool or a new one, performs a lazy connection if required.

async static putout(*amqp: DeliveredMessage*) → None

Send acknowledgement to broker with basic_client_ack

Parameters

amqp – pamqp.DeliveredMessage

async queue_length(*name: str*, ***options: Any*) → int

Get a queue length with queue_declare method.

Parameters

name – string, queue name

async send(*name: str*, *message: str*, *exchange: str = ''*, *properties: dict | None = None*, ***kwargs: Any*) → None

Raw message sending.

Parameters

- **name** – string, target queue name (routing_key)
- **message** – string, serialized message
- **exchange** – string, target exchange name
- **properties** – dict, for Basic.Properties
- ****kwargs** – dict, other basic_publish options

CHAPTER

NINE

KAFKA

Queue Broker based on kafka.

class microagent.tools.kafka.**KafkaBroker**(*dsn: str*, *uid: str = <factory>*, *log: ~logging.Logger = <Logger microagent.broker (WARNING)>*, *_bindings: dict[str, ~microagent.queue.Consumer] = <factory>*)

Experimental broker based on the Apache Kafka distributed stream processing system.

Parameters

- **dsn** – string, data source name for connection kafka://localhost:9092
- **log** – logging.Logger (optional)

Sending messages.

```
from microagent.tools.kafka import KafkaBroker

broker = KafkaBroker('kafka://localhost:9092')

await broker.user_created.send({'user_id': 1})
```

Consuming messages.

```
class EmailAgent(MicroAgent):
    @consumer(queues.mailer)
    async def example_read_queue(self, kafka, **data):
        # kafka: AIOKafkaConsumer
        process(data)
```

async send(*name: str*, *message: str*, ***kwargs: Any*) → None

Write a raw message to queue.

Parameters

- **name** – string, queue name
- **message** – string, serialized object
- ****kwargs** – specific parameters for each broker implementation

async bind(*name: str*) → None

Start reading queue.

Parameters

name – string, queue name

async queue_length(*name: str*, ***options: Any*) → int

Get the current queue length.

Parameters

- **name** – string, queue name
- ****options** – specific parameters for each broker implementation

CHAPTER

TEN

MOCKS & TESTING

Prepared bus and broker mocks for testing based on unittest.mock.AsyncMock

```
from microagent.tools.mocks import BusMock, BrokerMock

agent = Agent(bus=BusMock(), broker=BrokerMock())

agent.bus.user_created.send.assert_called()
agent.bus.user_created.call.assert_called()

agent.broker.mailing.send.assert_called()
agent.broker.mailing.length.assert_called()
```

class microagent.tools.mocks.**BusMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, *_spec_as_instance=False*, *_eat_self=None*, *unsafe=False*, ***kwargs*)

class microagent.tools.mocks.**BrokerMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, *_spec_as_instance=False*, *_eat_self=None*, *unsafe=False*, ***kwargs*)

CHAPTER

ELEVEN

AGENT EXAMPLES

user_agent.py

```
# mypy: ignore-errors
import os
from microagent import MicroAgent, on, periodic, receiver, cron, load_stuff

cur_dir = os.path.dirname(os.path.realpath(__file__))
signals, queues = load_stuff('file://' + os.path.join(cur_dir, 'signals.json'))


class UserAgent(MicroAgent):
    @on('pre_start')
    def setup(self):
        self.log.info('Run ...\n %s', self.info())

    @cron('* * * * *', timeout=5)
    async def example_cron_send_message(self):
        self.log.info('Run cron task')
        await self.broker.mailer.send({'text': 'Report text', 'email': 'admin@lwr.pw'})

    @periodic(period=15, timeout=10, start_after=3)
    async def example_periodic_task_send_signal(self):
        self.log.info('Run periodic task')
        await self.bus.user_comment.send('user_agent', user_id=1, text='informer text')

    @receiver(signals.user_created)
    async def example_signal_receiver_send_message(self, signal, sender, **kwargs):
        self.log.info('Catch signal %s from %s with %s', signal, sender, kwargs)
        await self.broker.mailer.send({'text': 'Welcome text', 'email': 'user@lwr.pw'})

    @receiver(signals.user_comment)
    async def example_rpc_call(self, **kwargs):
        self.log.info('Catch signal %s', kwargs)
        value = await self.bus.rpc_comments_count.call('user_agent', user_id=1)
        self.log.info('Get value = %s', value)
```

comment_agent.py

```
# mypy: ignore-errors
import os
import asyncio
```

(continues on next page)

(continued from previous page)

```
from collections import defaultdict
from microagent import MicroAgent, on, receiver, load_stuff

cur_dir = os.path.dirname(os.path.realpath(__file__))
signals, queues = load_stuff('file://' + os.path.join(cur_dir, 'signals.json'))


class CommentAgent(MicroAgent):
    @on('pre_start')
    async def setup(self):
        self.log.info('Run ...\n %s', self.info())
        self.comments_cache = defaultdict(lambda: 0)
        self.counter = 0

    @receiver(signals.rpc_comments_count)
    async def example_rpc_handler(self, user_id, **kwargs):
        self.log.info('Catch signal %s', kwargs)
        await asyncio.sleep(1)
        return self.comments_cache[user_id]

    @receiver(signals.user_comment)
    async def example_signal_receiver_send_message(self, user_id, **kwargs):
        self.log.info('Catch signal %s', kwargs)
        self.comments_cache[user_id] += 1
        await self.broker.mailer.send({'text': 'Comment', 'email': 'user@lwr.pw'})
        await self.broker.mailer.length()
```

email_agent.py

```
# mypy: ignore-errors
import os
from microagent import MicroAgent, on, consumer, load_stuff

cur_dir = os.path.dirname(os.path.realpath(__file__))
signals, queues = load_stuff('file://' + os.path.join(cur_dir, 'signals.json'))


class EmailAgent(MicroAgent):
    @on('pre_start')
    async def setup(self):
        self.log.info('Run ...\n %s', self.info())

    @consumer(queues.mailer)
    async def example_read_queue(self, **kwargs):
        self.log.info('Catch emailer %s', kwargs)
```

signals.json

```
{
    "version": 2,
    "signals": [
        {"name": "started", "providing_args": []},
```

(continues on next page)

(continued from previous page)

```
        {"name": "user_created", "providing_args": ["user_id"]},
        {"name": "user_comment", "providing_args": ["user_id"]},
        {"name": "rpc_comments_count", "providing_args": ["user_id"]},
        {"name": "comment_created", "providing_args": {
            "comment_id": ["number", "null"],
            "user_id": "number"
        }}
    ],
    "queues": [
        {"name": "mailer"}
    ]
}
```

CHAPTER

TWELVE

CONFIGURATION, LAUNCH AND ETC.

Configuration file of deploying unit, for shipped launcher (settings.py)

```
import sys
import logging

logging.basicConfig(format=(
    '%(levelname)-8s [pid#%(process)d] %(asctime)s %(name)s '
    '%(filename)s:%(lineno)d %(message)s'
), stream=sys.stdout, level=logging.DEBUG)


BUS = {
    'redis': {
        'backend': 'microagent.tools.redis.RedisSignalBus',
        'dsn': 'redis://localhost/7',
        'prefix': 'PREF',
    },
}

BROKER = {
    'redis': {
        'backend': 'microagent.tools.redis.RedisBroker',
        'dsn': 'redis://localhost/7',
    },
}


AGENT = {
    'user_agent': {
        'backend': 'examples.user_agent.UserAgent',
        'bus': 'redis',
        'broker': 'redis',
    },
    'comment_agent': {
        'backend': 'examples.comment_agent.CommentAgent',
        'bus': 'redis',
        'broker': 'redis',
    },
    'email_agent': {
        'backend': 'examples.email_agent.EmailAgent',
```

(continues on next page)

(continued from previous page)

```
        'broker': 'redis',
    },
}
```

Run in shell:

```
$ marun examples.settings
```

Custom server setup and starting (redis_server.py)

```
# mypy: ignore-errors
import sys
import asyncio
import logging
from microagent.tools.redis import RedisSignalBus, RedisBroker

from user_agent import UserAgent
from comment_agent import CommentAgent
from email_agent import EmailAgent

logging.basicConfig(format=(
    '%(levelname)-8s [pid#%(process)d] %(asctime)s %(name)s '
    '%(filename)s:%(lineno)d %(message)s'
), stream=sys.stdout, level=logging.DEBUG)


async def main():
    bus = RedisSignalBus('redis://localhost/7')
    broker = RedisBroker('redis://localhost/7')
    await bus.started.send('user_agent')

    user_agent = UserAgent(bus=bus, broker=broker)
    comment_agent = CommentAgent(bus=bus, broker=broker)
    email_agent = EmailAgent(broker=broker)

    await user_agent.start()
    await comment_agent.start()
    await email_agent.start()

    while True:
        await asyncio.sleep(60)


if __name__ == '__main__':
    asyncio.run(main())
```

Run in shell:

```
$ python examples/redis_server.py
```

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

G

I

K

L

M

N

O

P

Q

R

S

U

W