mango

Release 0.1

mango team

Sep 12, 2023

CONTENTS:

mango (modular python agent framework) is a python library for *multi-agent systems (MAS)*. It is written on top of asyncio and is releassed under the MIT license.

mango allows the user to create simple agents with little effort and in the same time offers options to structure agents with complex behaviour. The main features of mango are listed below.

CHAPTER

ONE

FEATURES

- Container mechanism to speedup local message exchange
- Message definition based on the FIPA ACL standard
- Structuring complex agents with loose coupling and agent roles
- Built-in codecs: JSON and protobuf
- Supports communication between agents directly via TCP or via an external MQTT broker in the middle

1.1 Installation

mango requires Python >= 3.8 and runs on Linux, OSX and Windows. For installation of mango you could use virtualenv which can create isolated Python environments for different projects.

It is also recommended to install virtualenvwrapper which makes it easier to manage different virtual environments.

1.1.1 Installation with pip

Once you have created a virtual environment you can just run pip to install it:

```
$ pip install mango-agents
```

1.1.2 Using a local message broker

If you want to make use of the functional mqtt modules to modularize your agent, you must have a local message broker running on your system. We recommend Mosquitto. On Ubuntu it can be installed as follows:

```
$ sudo apt-get install mosquitto
```

1.2 Getting started

In this section you will get to know the necessary steps to create a simple multi-agent system using *mango*. For an introduction to the different features that mango offers, we refer to the *Tutorial*.

1.2.1 Creating an agent

In our first example, we create a very simple agent that simply prints the content of all messages it receives:

```
from mango import Agent


class RepeatingAgent(Agent):
    def __init__(self, container):
        # We must pass a reference of the container to "mango.Agent":
        super().__init__(container)
        print(f"Hello world! My id is {self.aid}.")

    def handle_message(self, content, meta):
        # This method defines what the agent will do with incoming messages.
        print(f"Received a message with the following content: {content}")
```

Agents must be a subclass of `Agent`. This base class needs a reference to the container that the agents live in, so you must forward a *container* argument to it if you override `__init__()`.

1.2.2 Creating a container

Agents live in a container, so we need to know how to create a mango container. The container is responsible for message exchange between agents. More information about container and agents can be found in *Agents and container*

```
from mango import create_container
# Containers need to be started via a factory function.
# This method is a coroutine so it needs to be called from a coroutine using the
# await statement
async def get_container():
    return await create_container(addr=('localhost', 5555))
```

This is how a container is created. Since the method `create_container()` is a coroutine we need to await its result.

1.2.3 Running your first agent within a container

To put it all together we will wrap the creation of a container and the agent into a coroutine and execute it using `asyncio.run()`. The following script will create a RepeatingAgent and let it run within a container for three seconds and then shutdown the container:

```
import asyncio
from mango import Agent
from mango import create_container
```

(continues on next page)

(continued from previous page)

```
class RepeatingAgent(Agent):
    def __init__(self, container):
        # We must pass a ref. to the container to "mango.Agent":
        super().__init__(container)
        print(f"Hello world! My id is {self.aid}.")

    def handle_message(self, content, meta):
        # This method defines what the agent will do with incoming messages.
        print(f"Received a message with the following content: {content}")


async def run_container_and_agent(addr, duration):
    first_container = await create_container(addr=addr)
    first_agent = RepeatingAgent(first_container)
    await asyncio.sleep(duration)
    await first_container.shutdown()

asyncio.run(run_container_and_agent(addr=('localhost', 5555), duration=3))
```

The only output you should see is "Hello world! My id is agent0.", because the agent does not receive any other messages.

1.2.4 Creating a proactive Agent

Let's implement another agent that is able to send a hello world message to another agent:

```
from mango import Agent

    class HelloWorldAgent(Agent):
        def __init__(self, container, other_addr, other_id):
            super().__init__(container)
            self.schedule_instant_task(coroutine=self.context.send_acl_message(
                receiver_addr=other_addr,
                receiver_id=other_id,
                content="Hello world!")
            )

        def handle_messafe(self, content, meta):
            print(f"Received a message with the following content: {content}")
```

We are using the scheduling API, which is explained in further detail in the section *Scheduling and Clock*.

1.2.5 Connecting two agents

We can now connect an instance of a HelloWorldAgent with an instance of a RepeatingAgent and let them run.

```
import asyncio
from mango import Agent
from mango import create_container


class RepeatingAgent(Agent):
    def __init__(self, container):
        # We must pass a ref. to the container to "mango.Agent":
        super().__init__(container)
        print(f"Hello world! My id is {self.aid}.")

    def handle_message(self, content, meta):
        # This method defines what the agent will do with incoming messages.
        print(f"Received a message with the following content: {content}")


class HelloWorldAgent(Agent):
    def __init__(self, container, other_addr, other_id):
        super().__init__(container)
        self.schedule_instant_acl_message(
            receiver_addr=other_addr,
            receiver_id=other_id,
            content="Hello world!"
        )

    def handle_message(self, content, meta):
        print(f"Received a message with the following content: {content}")


async def run_container_and_two_agents(first_addr, second_addr):
    first_container = await create_container(addr=first_addr)
    second_container = await create_container(addr=second_addr)
    first_agent = RepeatingAgent(first_container)
    second_agent = HelloWorldAgent(second_container, first_container.addr, first_agent.
↪aid)
    await asyncio.sleep(1)
    await first_agent.shutdown()
    await second_agent.shutdown()
    await first_container.shutdown()
    await second_container.shutdown()


if __name__ == '__main__':
    asyncio.run(run_container_and_two_agents(
        first_addr=('localhost', 5555), second_addr=('localhost', 5556)))
```

You should now see the following output:

Hello world! My id is agent0. Received a message with the following content: Hello world!

You have now successfully created two agents and connected them.

1.3 Migration

Some mango versions will break the API; in this case, we may provide instructions on the migration on this page.

1.3.1 mango 0.4.0 to 1.0.0

- The module core and role have been restructured/moved: Agent, Container, all Role-related classes, and the container factory are now available using the top-level mango module, f.e. 'from mango import Agent, create_container'
- The methods 'handle_msg' in Agent and Role have been removed: Use 'handle_message' instead
- The parameters create_acl and acl_metadata from 'send_message' has been removed: use send_acl_message instead
- The parameter mqtt_kwargs from 'send_message' has been removed: use kwargs instead
- The DateTimeScheduledTask has been removed: use TimestampScheduledTask instead
- The context and scheduler of an agent are no longer public: use the convenience methods for sending/scheduling or _context, _scheduler from within the agent

1.4 Tutorial

1.4.1 Introduction

This tutorial gives an overview of the basic functions of mango agents and containers. It consists of four parts building a scenario of two PV plants, operated by their respective agents being directed by a remote controller.

Each part comes with a standalone executable file. Subsequent parts either extend the functionality or simplify some concept in the previous part.

As a whole, this tutorial covers:

- container and agent creation
- message passing within a container
- message passing between containers
- codecs
- scheduling
- roles

1.4.2 1. Setup and Message Passing

Corresponding file: *v1_basic_setup_and_message_passing.py*

For your first mango tutorial, you will learn the fundamentals of creating mango agents and containers as well as making them communicate with each other.

This example covers:

- container
- agent creation

- basic message passing
- clean shutdown of containers

First, we want to create two simple agents and have the container send a message to one of them. An agent is created by defining a class that inherits from the base Agent class of mango. Every agent must implement the `handle_message` method to which incoming messages are forwarded by the container.

```
from mango import Agent

class PVAgent(Agent):
    def __init__(self, container):
        super().__init__(container)
        print(f"Hello I am a PV agent! My id is {self.aid}.")

    def handle_message(self, content, meta):
        print(f"Received message with content: {content} and meta {meta}.")
```

Now we are ready to instantiate our system. mango is fundamentally built on top of asyncio and a lot of its functions are provided as coroutines. This means, practically every mango executable file will implement some variation of this pattern:

```
import asyncio

async def main():
    # do whatever here

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

First, we create the container. A container is created via the `mango.create_container` coroutine which requires at least the address of the container as a parameter.

```
PV_CONTAINER_ADDRESS = ("localhost", 5555)

# defaults to tcp connection
pv_container = await create_container(addr=PV_CONTAINER_ADDRESS)
```

Now we can create our agents. Agents always live inside a container and this container must be passed to their constructor.

```
# agents always live inside a container
pv_agent_0 = PVAgent(pv_container)
pv_agent_1 = PVAgent(pv_container)
```

For now, our agents are purely passive entities. To make them do something, we need to send them a message. Messages are passed by the container via the `send_message` function always at least expects some content and a target address. To send a message directly to an agent, we also need to provide its agent id which is set by the container when the agent is created.

```
 # we can now send a simple message to an agent and observe that it is received:
# Note that as of now agent IDs are set automatically as agent0, agent1, ... in order of␣
↪instantiation.
await pv_container.send_message(
    "Hello, this is a simple message.",
```

(continues on next page)

(continued from previous page)

```
    receiver_addr=PV_CONTAINER_ADDRESS,
    receiver_id="agent0",
)
```

Finally, you should always cleanly shut down your containers before your program terminates.

```
# don't forget to properly shut down containers at the end of your program
# otherwise you will get an asyncio.exceptions.CancelledError
await pv_container.shutdown()
```

This concludes the first part of our tutorial. If you run this code, you should receive the following output:

> Hello I am a PV agent! My id is agent0.
>
> Hello I am a PV agent! My id is agent1.
>
> Received message with content: Hello, this is a simple message. and meta {'network_protocol': 'tcp', 'priority': 0}.

1.4.3 2. Messaging between Containers

Corresponding file: *v2_inter_container_messaging_and_basic_functionality.py*

In the previous example, you learned how to create mango agents and containers and how to send basic messages between them. In this example, you expand upon this. We introduce a controller agent that asks the current feed_in of our PV agents and subsequently limits the output of both to their minimum.

This example covers:

- message passing between different containers
- basic task scheduling
- setting custom agent ids
- use of ACL metadata

First, we define our controller Agent. To ensure it can message the pv agents we pass that information directly to it in the constructor. The control agent will send out messages to each pv agent, await their replies and act according to that information. To handle this, we also add some control structures to the constructor that we will later use to keep track of which agents have already answered our messages. As an additional feature, we will make it possible to manually set the agent of our agents by.

```
class ControllerAgent(Agent):
    def __init__(self, container, known_agents, suggested_aid=None):
        super().__init__(container, suggested_aid=suggested_aid)
        self.known_agents = known_agents
        self.reported_feed_ins = []
        self.reported_acks = 0
        self.reports_done = None
        self.acks_done = None
```

Next, we set up its `handle_message` function. The controller needs to distinguish between two message types: The replies to feed_in requests and later the acknowledgements that a new maximum feed_in was set by a pv agent. We use the `performative` field of the ACL message to do this. We set the `performative` field to `inform` for feed_in replies and to `accept_proposal` for feed_in change acknowledgements. All messages between containers are expected to be ACL Messages (or implement the split_content_and_meta function). Since we do not want to create the full ACL object

ourselves every time, within this example we always use the convenience method `container.send_acl_message`, which also supports setting the acl metadata.

```
class ControllerAgent(Agent):
    """..."""

    def handle_message(self, content, meta):
        performative = meta['performative']
        if performative == Performatives.inform:
            # feed_in_reply message
            self.handle_feed_in_reply(content)
        elif performative == Performatives.accept_proposal:
            # set_max_ack message
            self.handle_set_max_ack()
        else:
            print(f"{self.aid}: Received an unexpected message  with content {content}␣
↪and meta {meta}")

    def handle_feed_in_reply(self, feed_in_value):
        self.reported_feed_ins.append(float(feed_in_value))
        if len(self.reported_feed_ins) == len(self.known_agents):
            if self.reports_done is not None:
                self.reports_done.set_result(True)

    def handle_set_max_ack(self):
        self.reported_acks += 1
        if self.reported_acks == len(self.known_agents):
            if self.acks_done is not None:
                self.acks_done.set_result(True)
```

We do the same for our PV agents. We will also enable user defined agent ids here.

```
class PVAgent(Agent):
    def __init__(self, container, suggested_aid=None):
        super().__init__(container, suggested_aid=suggested_aid)
        self.max_feed_in = -1

    def handle_message(self, content, meta):
        performative = meta["performative"]
        sender_addr = meta["sender_addr"]
        sender_id = meta["sender_id"]

        if performative == Performatives.request:
            # ask_feed_in message
            self.handle_ask_feed_in(sender_addr, sender_id)
        elif performative == Performatives.propose:
            # set_max_feed_in message
            self.handle_set_feed_in_max(content, sender_addr, sender_id)
        else:
            print(f"{self.aid}: Received an unexpected message with content {content}␣
↪and meta {meta}")
```

When a PV agent receives a request from the controller, it immediately answers. Note two important changes to the first example here: First, within our message handling methods we can not `await send_acl_message` directly because

handle_message is not a coroutine. Instead, we pass send_acl_message as a task to the scheduler to be executed at once via the schedule_instant_task method. Second, we set acl_meta to contain the typing information of our message.

```
class PVAgent(Agent):
    """..."""

    def handle_ask_feed_in(self, sender_addr, sender_id):
        reported_feed_in = PV_FEED_IN[self.aid]  # PV_FEED_IN must be defined at the top
        content = reported_feed_in

        acl_meta = {"sender_addr": self.context.addr, "sender_id": self.aid,
                    "performative": Performatives.inform}

        # Note, could be shortened using self.schedule_instant_acl_message
        self.schedule_instant_task(
            self.context.send_acl_message(
                content=content,
                receiver_addr=sender_addr,
                receiver_id=sender_id,
                acl_metadata=acl_meta,
            )
        )

    def handle_set_feed_in_max(self, max_feed_in, sender_addr, sender_id):
        self.max_feed_in = float(max_feed_in)
        print(f"{self.aid}: Limiting my feed_in to {max_feed_in}")
        self.schedule_instant_task(
            self.context.send_acl_message(
                content=None,
                receiver_addr=sender_addr,
                receiver_id=sender_id,
                acl_metadata={'performative': Performatives.accept_proposal},
            )
        )
```

Now both of our agents can handle their respective messages. The last thing to do is make the controller actually perform its active actions. We do this by implementing a run function with the following control flow: - send a feed_in request to each known pv agent - wait for all pv agents to answer - find the minimum reported feed_in - send a maximum feed_in setpoint of this minimum to each pv agent - again, wait for all pv agents to reply - terminate

```
class ControllerAgent(Agent):
    """..."""

    async def run(self):
        # we define an asyncio future to await replies from all known pv agents:
        self.reports_done = asyncio.Future()
        self.acks_done = asyncio.Future()

        # Note: For messages passed between different containers (i.e. over the network␣
↪socket) it is expected
        # that the message is an ACLMessage object. We can let the container wrap our␣
↪content in such an
        # object with using the send_acl_message method.
```

(continues on next page)

(continued from previous page)

```
        # We distinguish the types of messages we send by adding a type field to our␣
↪content.

        # ask pv agent feed-ins
        for addr, aid in self.known_agents:
            content = None
            acl_meta = {"sender_addr": self.context.addr, "sender_id": self.aid,
                        "performative": Performatives.request}
            # alternatively we could call send_acl_message here directly and await it
            self.schedule_instant_task(
                self.context.send_acl_message(
                    content=content,
                    receiver_addr=addr,
                    receiver_id=aid,
                    acl_metadata=acl_meta,
                )
            )

        # wait for both pv agents to answer
        await self.reports_done

        # limit both pv agents to the smaller ones feed-in
        print(f"{self.aid}: received feed_ins: {self.reported_feed_ins}")
        min_feed_in = min(self.reported_feed_ins)

        for addr, aid in self.known_agents:
            content = min_feed_in
            acl_meta = {"sender_addr": self.context.addr, "sender_id": self.aid,
                        "performative": Performatives.propose}

            # alternatively we could call send_acl_message here directly and await it
            self.schedule_instant_task(
                self.context.send_acl_message(
                    content=content,
                    receiver_addr=addr,
                    receiver_id=aid,
                    acl_metadata=acl_meta,
                )
            )

        # wait for both pv agents to acknowledge the change
        await self.acks_done
```

Lastly, we call all relevant instantiations and the run function within our main coroutine:

```
PV_CONTAINER_ADDRESS = ("localhost", 5555)
CONTROLLER_CONTAINER_ADDRESS = ("localhost", 5556)
PV_FEED_IN = {
    'PV Agent 0': 2.0,
    'PV Agent 1': 1.0,
}
```

(continues on next page)

(continued from previous page)

```
async def main():
    pv_container = await create_container(addr=PV_CONTAINER_ADDRESS)
    controller_container = await create_container(addr=CONTROLLER_CONTAINER_ADDRESS)

    # agents always live inside a container
    pv_agent_0 = PVAgent(pv_container, suggested_aid='PV Agent 0')
    pv_agent_1 = PVAgent(pv_container, suggested_aid='PV Agent 1')

    # We pass info of the pv agents addresses to the controller here directly.
    # In reality, we would use some kind of discovery mechanism for this.
    known_agents = [
        (PV_CONTAINER_ADDRESS, pv_agent_0.aid),
        (PV_CONTAINER_ADDRESS, pv_agent_1.aid),
    ]

    controller_agent = ControllerAgent(controller_container, known_agents, suggested_aid=
↪'Controller')

    # the only active component in this setup
    await controller_agent.run()

    # always properly shut down your containers
    await pv_container.shutdown()
    await controller_container.shutdown()

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

This concludes the second part of our tutorial. If you run this code you should receive the following output:

```
Controller: received feed_ins: [2.0, 1.0]
PV Agent 0: Limiting my feed_in to 1.0
PV Agent 1: Limiting my feed_in to 1.0
```

1.4.4 3. Using Codecs to simplify Message Types

Corresponding file: *v3_codecs_and_typing.py*

In example 2, you created some basic agent functionality and established inter-container communication. Message types were distinguished by the performative field of the meta information. This approach is tedious and prone to error. A better way is to use dedicated message objects and using their types to distinguish messages.

If instances of custom classes are exchanged over the network (or generally between different containers), these instances need to be serialized. In mango, objects can be encoded by mango's codecs. To make a new object type known to a codec it needs to provide a serialization and a deserialization method. The object type together with these methods is then passed to the codec which in turn is passed to a container. The container will then automatically use these methods when it encounters an object of this type as the content of a message.

This example covers:

- message classes
- codec basics
- the json_serializable decorator

We want to use the types of custom message objects as the new mechanism for message typing. We define these as simple data classes. For simple classes like this, we can use the `json_serializable` decorator to provide us with the serialization functionality.

```
import mango.messages.codecs as codecs
from dataclasses import dataclass

@codecs.json_serializable
@dataclass
class AskFeedInMsg:
    pass


@codecs.json_serializable
@dataclass
class FeedInReplyMsg:
    feed_in: int


@codecs.json_serializable
@dataclass
class SetMaxFeedInMsg:
    max_feed_in: int


@codecs.json_serializable
@dataclass
class MaxFeedInAck:
    pass
```

Next, we need to create a codec, make our message objects known to it, and pass it to our containers.

```
my_codec = codecs.JSON()
my_codec.add_serializer(*AskFeedInMsg.__serializer__())
my_codec.add_serializer(*SetMaxFeedInMsg.__serializer__())
my_codec.add_serializer(*FeedInReplyMsg.__serializer__())
my_codec.add_serializer(*MaxFeedInAck.__serializer__())

pv_container = await create_container(addr=PV_CONTAINER_ADDRESS, codec=my_codec)

controller_container = await create_container(
    addr=CONTROLLER_CONTAINER_ADDRESS, codec=my_codec
)
```

Any time the content of a message matches one of these types now the corresponding serialize and deserialize functions are called. Of course, you can also create your own serialization and deserialization functions with more sophisticated behaviours and pass them to the codec. For more details refer to the `codecs` section of the documentation.

With this, the message handling in our agent classes can be simplified:

```
class ControllerAgent(Agent):
    """..."""

    def handle_message(self, content, meta):
```

(continues on next page)

(continued from previous page)

```
        if isinstance(content, FeedInReplyMsg):
            self.handle_feed_in_reply(content.feed_in)
        elif isinstance(content, MaxFeedInAck):
            self.handle_set_max_ack()
        else:
            print(f"{self.aid}: Received a message of unknown type {type(content)}")


class PVAgent(Agent):
    """..."""

    def handle_message(self, content, meta):
        sender_addr = meta["sender_addr"]
        sender_id = meta["sender_id"]

        if isinstance(content, AskFeedInMsg):
            self.handle_ask_feed_in(sender_addr, sender_id)
        elif isinstance(content, SetMaxFeedInMsg):
            self.handle_set_feed_in_max(content.max_feed_in, sender_addr, sender_id)
        else:
            print(f"{self.aid}: Received a message of unknown type {type(content)}")
```

This concludes the third part of our tutorial. If you run the code, you should receive the same output as in part 2:

```
Controller: received feed_ins: [2.0, 1.0]
PV Agent 0: Limiting my feed_in to 1.0
PV Agent 1: Limiting my feed_in to 1.0
```

1.4.5 4. Scheduling and Roles

Corresponding file: *v4_scheduling_and_roles.py*

In example 3, you restructured your code to use codecs for easier handling of typed message objects. Now it is time to expand the functionality of our controller. In addition to setting the maximum feed_in of the pv agents, the controller should now also periodically check if the pv agents are still reachable.

To achieve this, the controller should send a regular "ping" message to each pv agent that is in turn answered by a corresponding "pong". Periodic tasks can be handled for you by mango's scheduling API.

With the introduction of this task, we know have different responsibilities for the agents (e. g. act as PVAgent and reply to ping requests). In order to facilitate structuring an agent with different responsibilities we can use the role API. The idea of using roles is to divide the functionality of an agent by responsibility in a structured way.

A role is a python object that can be assigned to a RoleAgent. The two main functions each role implements are:

- __init__ - where you do the initial object setup
- setup - which is called when the role is assigned to an agent

This distinction is relevant because only within *setup* the RoleContext (i.e. access to the parent agent and container) exist. Thus, things like message handlers that require container knowledge are introduced there.

This example covers:

- role API basics

- scheduling and periodic tasks

The key part of defining roles are their `__init__` and `setup` methods. The first is called to create the role object. The second is called when the role is assigned to an agent. In our case, the main change is that the previous distinction of message types within `handle_message` is now done by subscribing to the corresponding message type to tell the agent it should forward these messages to this role. The `subscribe_message` method expects, besides the role and a handle method, a message condition function. The idea of the condition function is to allow to define a condition filtering incoming messages. Another idea is that sending messages from the role is now done via its context with the method: `self.context.send_acl_message`.

We first create the `Ping` role, which has to periodically send out its messages. We can use mango's scheduling API to handle this for us via the `schedule_periodic_tasks` function. This takes a coroutine to execute and a time interval. Whenever the time interval runs out the coroutine is triggered. With the scheduling API you can also run tasks at specific times. For a full overview we refer to the documentation.

```
from mango import Role

class PingRole(Role):
    def __init__(self, ping_recipients, time_between_pings):
        self.ping_recipients = ping_recipients
        self.time_between_pings = time_between_pings
        self.ping_counter = 0
        self.expected_pongs = []

    def setup(self):
        self.context.subscribe_message(
            self, self.handle_pong, lambda content, meta: isinstance(content, Pong)
        )

        # this task is automatically executed every "time_between_pings" seconds
        self.context.schedule_periodic_task(self.send_pings, self.time_between_pings)

    async def send_pings(self):
        for addr, aid in self.ping_recipients:
            ping_id = self.ping_counter
            msg = Ping(ping_id)
            meta = {"sender_addr": self.context.addr, "sender_id": self.context.aid}

            await self.context.send_acl_message(
                msg,
                receiver_addr=addr,
                receiver_id=aid,
                acl_metadata=meta,
            )
            self.expected_pongs.append(ping_id)
            self.ping_counter += 1

    def handle_pong(self, content, meta):
        if content.pong_id in self.expected_pongs:
            print(
                f"Pong {self.context.aid}: Received an expected pong with ID: {content.
↪pong_id}"
            )
            self.expected_pongs.remove(content.pong_id)
```

(continues on next page)

(continued from previous page)

```
        else:
            print(
                f"Pong {self.context.aid}: Received an unexpected pong with ID: {content.
↪pong_id}"
            )
```

The ControllerRole now covers the former responsibilities of the controller:

```
class ControllerRole(Role):
def __init__(self, known_agents):
    super().__init__()
    self.known_agents = known_agents
    self.reported_feed_ins = []
    self.reported_acks = 0
    self.reports_done = None
    self.acks_done = None

def setup(self):
    self.context.subscribe_message(
        self,
        self.handle_feed_in_reply,
        lambda content, meta: isinstance(content, FeedInReplyMsg),
    )

    self.context.subscribe_message(
        self,
        self.handle_set_max_ack,
        lambda content, meta: isinstance(content, MaxFeedInAck),
    )

    self.context.schedule_instant_task(self.run())
```

The methods `handle_feed_in_reply`, `handle_set_max_ack` and `run` are also part of this role and remain unchanged.

The `Pong` role is associated with the PV Agents and purely reactive.

```
class PongRole(Role):
    def setup(self):
        self.context.subscribe_message(
            self, self.handle_ping, lambda content, meta: isinstance(content, Ping)
        )

    def handle_ping(self, content, meta):
        ping_id = content.ping_id
        sender_addr = meta["sender_addr"]
        sender_id = meta["sender_id"]
        answer = Pong(ping_id)

        print(f"Ping {self.context.aid}: Received a ping with ID: {ping_id}")

        # message sending from roles is done via the RoleContext
        self.context.schedule_instant_task(
```

(continues on next page)

(continued from previous page)

```
            self.context.send_acl_message(
                answer,
                receiver_addr=sender_addr,
                receiver_id=sender_id,
            )
        )
```

Since the PV Agent is purely reactive, its other functionality stays basically unchanged and is simply moved to the PVRole.

```
class PVRole(Role):
    def __init__(self):
        self.max_feed_in = -1

    def setup(self):
        self.context.subscribe_message(
            self,
            self.handle_ask_feed_in,
            lambda content, meta: isinstance(content, AskFeedInMsg),
        )
        self.context.subscribe_message(
            self,
            self.handle_set_feed_in_max,
            lambda content, meta: isinstance(content, SetMaxFeedInMsg),
        )

    def handle_ask_feed_in(self, content, meta):
        """..."""
    self.context.schedule_instant_task(
        self.context.send_acl_message(
            content=msg,
            receiver_addr=sender_addr,
            receiver_id=sender_id,
        )
    )

    def handle_set_feed_in_max(self, content, meta):
        """..."""
        self.context.schedule_instant_task(
            self.context.send_acl_message(
                content=msg,
                receiver_addr=sender_addr,
                receiver_id=sender_id,
            )
        )
```

The definition of the agent classes itself now simply boils down to assigning it all the roles it has:

```
from mango import RoleAgent

class PVAgent(RoleAgent):
    def __init__(self, container):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(container)
        self.add_role(PongRole())
        self.add_role(PVRole())

class ControllerAgent(RoleAgent):
    def __init__(self, container, known_agents):
        super().__init__(container)
        self.add_role(PingRole(known_agents, 2))
        self.add_role(ControllerRole(known_agents))
```

This concludes the last part of our tutorial. If you want to run the code, you don't need to await the run method of the controller anymore, since everything now happens automatically within the roles. In your `main`, you can replace the line:

```
await controller_agent.run()
```

with the following line:

```
await asyncio.sleep(5)
```

If you then run this code, you should receive the following output:

```
Ping PV Agent 0: Received a ping with ID: 0
Ping PV Agent 1: Received a ping with ID: 1
Pong Controller: Received an expected pong with ID: 0
Pong Controller: Received an expected pong with ID: 1
Controller received feed_ins: [2.0, 1.0]
PV Agent 0: Limiting my feed_in to 1.0
PV Agent 1: Limiting my feed_in to 1.0
Ping PV Agent 0: Received a ping with ID: 2
Ping PV Agent 1: Received a ping with ID: 3
Pong Controller: Received an expected pong with ID: 2
Pong Controller: Received an expected pong with ID: 3
Ping PV Agent 0: Received a ping with ID: 4
Ping PV Agent 1: Received a ping with ID: 5
Pong Controller: Received an expected pong with ID: 4
Pong Controller: Received an expected pong with ID: 5
```

1.5 Agents and container

1.5.1 mango container

In mango, agents live in a `container`. The container is responsible for everything network related of the agent. This includes in particular sending and receiving of messages, but also message distribution to the correct agent or (de-)serialization of messages. Container also help to to speed up message exchange between agents that run on the same physical hardware, as data that is exchanged between such agents will not have to be sent through the network.

In mango, a container is created using the classmethod `mango.create_container`:

```
@classmethod
async def create_container(cls, *, connection_type: str = 'tcp', codec: Codec = None,␣
↪clock: Clock = None,
                   addr: Optional[Union[str, Tuple[str, int]]] = None,
                   proto_msgs_module=None,
                   **kwargs):
```

The factory method is a coroutine, so it has to be scheduled within a running asyncio loop. A simple container, that uses plain tcp for message exchange can be created as follows:

```
import asyncio
from mango import create_container

async def get_simple_container():
    container = await create_container(addr=('localhost', 5555))
    return container

simple_container = asyncio.run(get_simple_container())
```

A container can be parametrized regarding its connection type ('tcp' or 'MQTT') and regarding the codec that is used for message serialization. The default codec is JSON (see section *Codecs* for more information). It is also possible to define the clock that an agents scheduler should use (see section scheduling).

After a container is created, it is waiting for incoming messages on the given address. As soon as the container has some agents, it will distribute incoming messages to the corresponding agents and allow agents to send messages to other agents.

At the end of its lifetime, a `container` should be shutdown by using the method `shutdown()`. It will then shutdown all agents that are still running in this container and cancel running tasks.

1.5.2 mango agents

mango agents can be implemented by inheriting from the abstract class `mango.Agent`. This class provides basic functionality such as to register the agent at the container or to constantly check the inbox for incoming messages. Every agent lives in exactly one container and therefore an instance of a container has to be provided when `__init__()` of an agent is called. Custom agents that inherit from the `Agent` class have to call `super().__init__(container, suggested_aid: str = None)__` on initialization. This will register the agent at the provided container instance and will assign a unique agent id (`self.aid`) to the agent. However, it is possible to suggest an aid by setting the variable `suggested_aid` to your aid wish. The aid is granted if there is no other agent with this id, and if the aid doesn't interfere with the default aid pattern, otherwise the generated aid will be used. To check if the aid is available beforehand, you can use `container.is_aid_available`. It will also create the task to check for incoming messages.

1.5.3 agent process

To improve multicore utilization, mango provides a way to distribute agents to processes. For this, it is necessary to create and register the agent in a slightly different way.

The process_handle is awaitable and will finish exactly when the process is fully set up. Further, it contains the pid *process_handle.pid*.

Note that after the creation, the agent lives in a mirror container in another process. Therefore, it is not possible to interact with the agent directly from the main process. If you want to interact with the agent after the creation, it is possible to dispatch a task in the agent process using *dispatch_to_agent_process*.

1.6 Message exchange

1.6.1 Receiving messages

Custom agents that inherit from the `Agent` class are able to receive messages from other agents via the method `handle_message`. Hence this method has to be overwritten. The structure of this method looks like this:

```
@abstractmethod
def handle_message(self, content, meta: Dict[str, Any]):

    raise NotImplementedError
```

Once a message arrives at a container, the container is responsible to deserialize the message and to split the content from all meta information. While the meta information may include e. g. information about the sender of the message or about the performative, the content parameter holds the actual content of the message.

A simple agent, that just prints the content and meta information of incoming messages could look like this:

```
from mango import Agent

class SimpleReceivingAgent(Agent):
    def __init__(self, container):
        super().__init__(container)

    def handle_message(self, content, meta):
        print(f'{self.aid} received a message with content {content} and'
            f'meta {meta}')
```

1.6.2 Sending messages

Agents are able to send messages to other agents via the container method send_message:

```
async def send_message(self, content,
                       receiver_addr: Union[str, Tuple[str, int]], *,
                       receiver_id: Optional[str] = None,
                       **kwargs) -> bool:
```

To send a tcp message, the receiver address and receiver id (the agent id of the receiving agent) has to be provided. *content* defines the content of the message. This will appear as the *content* argument at the receivers handle_message() method.

If you want to send an ACL-message use the method `container.send_acl_message`, which will wrap the content in a ACLMessage using `create_acl` internally.

```
async def send_acl_message(self, content,
                       receiver_addr: Union[str, Tuple[str, int]], *,
                       receiver_id: Optional[str] = None,
                       acl_metadata: Optional[Dict[str, Any]] = None,
                       **kwargs) -> bool:
```

The argument `acl_metadata` enables to set all meta information of an acl message. It expects a dictionary with the field name as string as a key and the field value as key. For example:

```
from mango.messages.message import Performatives

example_acl_metadata = {
    'performative': Performatives.inform,
    'sender_id': 'agent0',
    'sender_addr': ('localhost', 5555),
    'conversation_id': 'conversation01'
}
```

The argument `kwargs` can be used to set specific configs, if the container is connected via MQTT to a message broker.

1.7 Scheduling and Clock

When implementing agents including proactive behavior there are some typical types of tasks you might want to create. For example it might be desired to let the agent check every minute whether some resources are available, or often you just want to execute a task at a specified time. To help achieving this kind of goals mango exposes the scheduling API.

The core of this API is the scheduler, which is part of every agent. To schedule a task its necessary to create a ScheduledTask (resp. an object of a subclass). In mango the following tasks are available:

Table 1: Available ScheduledTasks

Class	Description
InstantScheduledTask	Executes the coroutine without delay
TimestampScheduledTask	Executes the coroutine at a specified datetime
PeriodicScheduledTask	Executes a coroutine periodically with a static delay between the cycles
ConditionalScheduledTask	Executes the coroutine when a specified condition evaluates to True

Furthermore there are convenience methods to get rid of the class imports when using these types of tasks.

```
from mango import Agent
from mango.util.scheduling import InstantScheduledTask

    class ScheduleAgent(Agent):
        def __init__(self, container, other_addr, other_id):
            self.schedule_instant_acl_message(
                receiver_addr=other_addr,
                receiver_id=other_id,
                content="Hello world!")
            )
            # equivalent to
            self.schedule_instant_acl_message(
                receiver_addr=other_addr,
                receiver_id=other_id,
                content="Hello world!"))
            )

        def handle_message(self, content, meta: Dict[str, Any]):
            pass
```

When using the scheduling another feature becomes available: suspendable tasks. This makes it possible to pause and resume all tasks started with the scheduling API. Using this it is necessary to specify an identifier when starting the task

(using *src=your_identifier*). To suspend a task you can call *scheduler.suspend(your_identifier)*, to resume them just call the counterpart *scheduler.resume(your_identifier)*. The scheduler is part of the agent and accessible via *self._scheduler*.

1.7.1 Dispatch Tasks to other Process

As asyncio does not provide real parallelism to utilize multiple cores and agents may have tasks, which need a lot computational power, the need to dispatch certain tasks to other processes appear. Handling inter process communication manually is quite exhausting and having multiple process pools across different roles or agents leads to inefficient resource allocations. As a result mango offers a way to dispatch tasks, based on coroutine-functions, to other processes, managed by the framework.

Analogues to the normal API there are two different ways, first you create a ScheduledProcessTask and call `schedule_process_task`, second you invoke the convnience methods with "process" in the name. These methods exists on any Agent, the RoleContext and the Scheduler.

```
from mango import Agent
from mango.util.scheduling import InstantScheduledProcessTask

    class ScheduleAgent(Agent):
        def __init__(self, container, other_addr, other_id):
            self.schedule_instant_process_task(coroutine_creator=lambda: self.context.
↪send_acl_message(
                receiver_addr=other_addr,
                receiver_id=other_id,
                content="Hello world!")
            )
            # equivalent to
            self.schedule_process_task(InstantScheduledProcessTask(coroutine_
↪creator=lambda: self.context.send_acl_message(
                receiver_addr=other_addr,
                receiver_id=other_id,
                content="Hello world!"))
            )

        def handle_message(self, content, meta: Dict[str, Any]):
            pass
```

1.7.2 Using an external clock

Usually, the scheduler will schedule the tasks of a mango agent based on the real time. This is the default behaviour of the scheduler. However, in some contexts it is necessary to schedule the agent based on an external clock, e. g. in simulations that run faster than real-time. In mango, this is possible by defining the `Clock` of a container, which will be used by the scheduler of all agents within this container. The default clock is the `AsyncioClock`, which works as a real-time clock. An alternative clock is the `ExternalClock`. Time of this clock has to be set by an external process. That way you can control how fast or slow time passes within your agent system:

```
import asyncio
from mango import create_container
from mango import Agent
from mango.util.clock import AsyncioClock, ExternalClock
```

(continues on next page)

(continued from previous page)

```
class Caller(Agent):
    def __init__(self, container, receiver_addr, receiver_id):
        super().__init__(container)
        self.schedule_timestamp_task(coroutine=self.send_hello_world(receiver_addr,␣
↪receiver_id),
                                     timestamp=self.current_timestamp + 5)

    async def send_hello_world(self, receiver_addr, receiver_id):
        await self.context.send_acl_message(receiver_addr=receiver_addr,
                                            receiver_id=receiver_id,
                                            content='Hello World')

    def handle_message(self, content, meta):
        pass


class Receiver(Agent):
    def __init__(self, container):
        super().__init__(container)
        self.wait_for_reply = asyncio.Future()

    def handle_message(self, content, meta):
        print(f'Received a message with the following content {content}.')
        self.wait_for_reply.set_result(True)


async def main():
    clock = AsyncioClock()
    # clock = ExternalClock(start_time=1000)
    addr = ('127.0.0.1', 5555)
    c = await create_container(addr=addr, clock=clock)
    receiver = Receiver(c)
    caller = Caller(c, addr, receiver.aid)
    await receiver.wait_for_reply
    await c.shutdown()


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

This code will terminate after 5 seconds. If you change the clock to an `ExternalClock` by uncommenting the ExternalClock in the example above, the program won't terminate as the time of the clock is not proceeded by an external process. If you comment in the ExternalClock and change your main() as follows, the program will terminate after one second:

```
async def main():
    # clock = AsyncioClock()
    clock = ExternalClock(start_time=1000)
    addr = ('127.0.0.1', 5555)

    c = await create_container(addr=addr, clock=clock)
    receiver = Receiver(c)
```

(continues on next page)

(continued from previous page)

```
    caller = Caller(c, addr, receiver.aid)
    if isinstance(clock, ExternalClock):
        await asyncio.sleep(1)
        clock.set_time(clock.time + 5)
    await receiver.wait_for_reply
    await c.shutdown()
```

1.8 Codecs

Most of the codec related code is taken and adapted from aiomas: https://gitlab.com/sscherfke/aiomas/

Codecs enable the container to encode and decode known data types to send them as messages. Mango already contains two codecs: A json serializer that can (recursively) handle any json serializable object and a protobuf codec that will wrap an object into a generic protobuf message. Other codecs can be implemented by inheriting from the `Codec` base class and implementing its `encode` and `decode` methods. Codecs will only handle types explicitely known to them. New known types can be added to a codec with the `add_serializer` method. This method expects a type together with a serialization method and a deserialization method that translate the object into a format the codec can handle (for example a json-serializable string for the json codec).

Warning: When using the json codec certain types can not be exactly serialized and deserialized between containers. One example are `tuple` and classes derived from it like `namedtuple`. The core of the json codec uses pythons json encoder [1] for any type that this encoder can handle by itself. Tuples are translated to json arrays without any further information by this encoder. Consequently, a receiving container will only see a json array and deserialize it to a python list.

[1]: https://docs.python.org/3/library/json.html#json.JSONEncoder

1.8.1 Quickstart

general use

Consider a simple example class we wish to encode as json:

```
class MyClass:
    def __init__(self, x, y):
        self.x = x
        self._y = y

    @property
    def y(self):
        return self._y

    def __asdict__(self):
        return {"x": self.x, "y": self.y}

    @classmethod
    def __fromdict__(cls, attrs):
        return cls(**attrs)
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def __serializer__(cls):
        return (cls, cls.__asdict__, cls.__fromdict__)
```

If we try to encode an object of `MyClass` without adding a serializer we get an SerializationError:

```
codec = codecs.JSON()

my_object = MyClass("abc", 123)
encoded = codec.encode(my_object)
```

```
python main.py
...
mango.messages.codecs.SerializationError: No serializer found for type "<class '__main__.
↪MyClass'>"
```

We have to make the type known to the codec to use it:

```
codec = codecs.JSON()
codec.add_serializer(*MyClass.__serializer__())

my_object = MyClass("abc", 123)
encoded = codec.encode(my_object)
decoded = codec.decode(encoded)

print(my_object.x, my_object.y)
print(decoded.x, decoded.y)
```

```
python main.py
abc 123
abc 123
```

All that is left to do now is to pass our codec to the container. This is done during container creation in the `create_container` method.

```
class SimpleReceivingAgent(Agent):
    def __init__(self, container):
        super().__init__(container)

    def handle_message(self, content, meta):
        print(f"{self.aid} received a message with content {content} and meta f{meta}")
        if isinstance(content, MyClass):
            print(content.x)
            print(content.y)


async def main():
    codec = codecs.JSON()
    codec.add_serializer(*MyClass.__serializer__())

    # codecs can be passed directly to the container
    # if no codec is passed a new instance of JSON() is created
```

(continues on next page)

(continued from previous page)

```
    sending_container = await create_container(addr=("localhost", 5556), codec=codec)
    receiving_container = await create_container(addr=("localhost", 5555), codec=codec)
    receiving_agent = SimpleReceivingAgent(receiving_container)

    # agents can now directly pass content of type MyClass to each other
    my_object = MyClass("abc", 123)
    await sending_container.send_acl_message(
        content=my_object, receiver_addr=("localhost", 5555), receiver_id="agent0"
    )

    await receiving_container.shutdown()
    await sending_container.shutdown()


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

```
python main.py
agent0 received a message with content <__main__.MyClass object at 0x7f42c930edc0> and␣
↪meta f{'sender_id': None, 'sender_addr': ['localhost', 5556], 'receiver_id': 'agent0',
↪'receiver_addr': ['localhost', 5555], 'performative': None, 'conversation_id': None,
↪'reply_by': None, 'in_reply_to': None, 'protocol': None, 'language': None, 'encoding':␣
↪None, 'ontology': None, 'reply_with': None, 'network_protocol': 'tcp', 'priority': 0}
abc
123
```

@json_serializable decorator

In the above example we explicitely defined methods to (de)serialize our class. For simple classes, especially data classes, we can achieve the same result (for json codecs) via the `@json_serializable` decorator. This creates the `__asdict__`, `__fromdict__` and `__serializer__` functions in the class:

```
from mango.messages.codecs import serializable

@json_serializable
class DecoratorData:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

def main():
    codec = codecs.JSON()
    codec.add_serializer(*DecoratorData.__serializer__())

    my_data = DecoratorData(1,2,3)
    encoded = codec.encode(my_data)
    decoded = codec.decode(encoded)

    print(my_data.x, my_data.y, my_data.z)
    print(decoded.x, decoded.y, decoded.z)
```

```
python main.py
1 2 3
1 2 3
```

1.8.2 fast json

Besides the normal full features json codec, which is able to serialize and deserialize messages under preservation of the type information, mango provides the *codecs.FastJson* codec. This codec usese *msgspec* and does not provide any type safety. Therefore are also no custom serializer.

1.8.3 proto codec and ACLMessage

Serialization methods for the proto codec are expected to encode the object into a protobuf message object with the `SerializeToString` method. The codec then wraps the message into a generic message wrapper, containing the serialized protobuf message object and a type id. This is necessary because in general the original type of a protobuf message can not be infered from its serialized form.

The `ACLMessage` class is encouraged to be used for fipa compliant agent communication. For ease of use it gets specially handled in the protobuf codec: Its content field may contain any proto object known to the codec and gets encoded with the associated type id just like a non-ACL message would be encoded into the generic message wrapper.

Here is an example class implementing a proto serializer for a proto message containing the same fields as the example class:

```
from msg_pb2 import MyOtherMsg
from mango.messages.message import ACLMessage

class SomeOtherClass:
    def __init__(self, x=1, y='abc', z=None) -> None:
        self.x = x
        self.y = y
        if z is None:
            self.z = {}
        else:
            self.z = z

    def __toproto__(self):
        msg = MyOtherMsg()
        msg.x = self.x
        msg.y = self.y
        msg.z = str(self.z)
        return msg

    @classmethod
    def __fromproto__(cls, data):
        msg = MyOtherMsg()
        msg.ParseFromString(data)
        return cls(msg.x, msg.y, eval(msg.z))

    @classmethod
    def __protoserializer__(cls):
        return cls, cls.__toproto__, cls.__fromproto__
```

(continues on next page)

(continued from previous page)

```
def main():
    codec = codecs.PROTOBUF()
    codec.add_serializer(*SomeOtherClass.__protoserializer__())

    my_object = SomeOtherClass()
    decoded = codec.decode(codec.encode(my_object))

    wrapper = ACLMessage()
    wrapper.content = my_object
    w_decoded = codec.decode(codec.encode(wrapper))

    print(my_object.x, my_object.y, my_object.z)
    print(decoded.x, decoded.y, decoded.z)
    print(
        wrapper_decoded.content.x,
        wrapper_decoded.content.y,
        wrapper_decoded.content.z,
    )
```

```
python main.py
1 2 abc123 {1: 'test', 2: 'data', 3: 123}
1 2 abc123 {1: 'test', 2: 'data', 3: 123}
1 2 abc123 {1: 'test', 2: 'data', 3: 123}
```

In case you want to directly pass proto objects as content to the codec (or as content to the containers `send_message`) you can shorten this process by making the proto type known to the codec using the `register_proto_type` function as in this example:

```
from msg_pb2 import MyMsg

def main():
codec = codecs.PROTOBUF()
codec.register_proto_type(MyMsg)

my_obj = MyMsg()
my_obj.content = b"some_bytes"
encoded = codec.encode(my_obj)
decoded = codec.decode(encoded)

print(my_obj)
print(encoded)
print(decoded)
```

```
python main.py
content: "some_bytes"

b'\x08\x01\x12\x0c\x12\nsome_bytes'
content: "some_bytes"
```

1.9 Role-API

Besides inheriting from the `Agent`-class there is another option to integrate features into an agent: the role API. The idea of using roles is to divide the functionality of an agent by responsibility in a structured way. The target of this API is to increase the reusability and the maintainability of the agents components. To achieve this the role API works with orchestration rather than inheriting to extend the agents features.

1.9.1 The RoleContext

The role context is the API to the environment of the role. It provides functionality to interact with other roles, to send messages to other agents or simply to fetch some meta data.

1.9.2 The Role

To implement a role you have to extend the abstract class `mango.Role`. Concrete instances of implementations can be assigned to the general `mango.RoleAgent`.

Lifecycle

The first step in the roles life is the instantiation via `__init__`. This is done by the user itself and can be used to configure the roles behavior. The next step is adding the role to a `RoleAgent` using `add_role`. The role will get notified by this through the method `setup`. After adding the role the `RoleContext` is available, which represents the environment (container, agent, other roles). When the role or agent got removed, or the container shut down, the hook-method `on_stop` will be called, so you can do some cleanup or send last messages before the life ends.

Note: After a shutdown or removal a role is **not** supposed to be reused! When you want to deactivate a role temporarily use the methods `activate` and `deactivate` of the RoleContext.

Sharing Data

There are two possible was to share data between the roles.

1. Using the data container in the RoleContext (`RoleContext.data`)
2. Creating explicit models using the model API of the RoleContext(`RoleContext.get_or_create_model`)

The first way is pretty straightforward. For example:

```
...
class MyRole(Role):
    def setup(self):
        self.context.data.my_item = "hello"
```

The stored entry `my_item` can be used in every other role of the same agent now.

The second way needs a bit more preparations. First we need to define a model as python class. The class object will be used as key, so every model-type can be stored exactly once.

```
class MyModel:
    def __init__(self):
        self.my_item = ""
...
class MyRole(Role):
    def setup(self):
        mymodel = self.context.get_or_create_model(MyModel)
        mymodel.my_item = 'hello'
```

One advantage of this approach is that a model is subscribable using the method `RoleContext.subscribe_model`. To make use of this every time the models changed `RoleContext.update` has to be called.

Handle Messages

As in a normal agent implementation, roles can handle incoming messages. To add a message handler you can use `RoleContext.subscribe_message`. This method expects, besides the role and a handle method, a message condition function. The handle method must have exactly two arguments (excl. `self`) `content` and `meta`. The condition function must have exactly one argument `content`. The idea of the condition function is to allow to define a condition filtering incoming messages, so you only handle one type of message per handler. Furthermore you can define a `priority` of the message subscription, this will be used to determine the message dispatch order (lower number = earlier execution, default=0).

```
...
class MyRole(Role):
    def setup(self):
        self.context.subscribe_message(self, self.handle_ping, lambda content:␣
↪isinstance(content, Ping))

    def handle_ping(self, content, meta):
        print('Ping received!')
```

Deactivate/Activate other Roles

Sometimes you might want to deactivate the functionality of a whole role, for example when you entered a new coalition you don't want to accept new coalition invites. It would of course be possible to manage this case with shared data and controlling flags, but this requires a lot of additional code and might lead to errors when implementing it. Furthermore, it increases the complexity of the implemented roles. To tackle this scenario a native deactivation/activation of roles is possible in mango. To deactivate a role the method `RoleContext.deactivate` can be used. To activate it again, use `RoleContext.activate`. When a role is deactivated

1. it is not possible to handle messages anymore
2. the role will not get updates on shared models anymore
3. all scheduled tasks get suspended.

When a role activated again all three point are completely reverted.

Note: Suspending of tasks might not work immediately, as it intercepts `__await__`.

1.10 Development Guidelines

As we mainly work in a critical domain, we set great value on code quality not only to ensure correctness, but also to improve readability and maintainability. To reach this goal we have to set some standards regarding the development process and the test quality.

1.10.1 Quickstart

In mango it is not possible to directly push on the branches *development* or *master*. Both branches are protected and changes can only be merged using a gitlab merge-request. So when you work on a feature, the typical process would be to create a feature-branch. When you are finished you just have to create a merge-request, pass the CI/CD pipeline, make a maintainer review the changes, and you are ready to merge!

1.10.2 CI/CD

To monitor the quality, issues found by linters, code coverage and correctness of the tests, we need an automated process triggered on every branch for every commit. That is done by our CI/CD pipeline. You can find the code-coverage, linting and the test-reports there.

Continuous deployment to PyPi is planned but not ready yet.

1.10.3 Quality guidelines

Tests

We understand testing as part of the normal software development process. That leads to writing automated tests for every new feature, every fixed bug. Furthermore, it is necessary to ensure that we test our code integratively **and** with unit tests. As a consequence every feature have to be part of an integration test and should have its own unit tests additionally. Doing that we can be sure that the code is working correctly when integrated in a typical use case and when executed "standalone".

Unit Tests

A unit test is a test for the smallest possible testable part of the code. This is often a method or a class, in case of mango it can be a bit bigger though, because asyncio is heavily used.

Integration Tests

An integration test is everything what aims to test more than one unit.

Coverage

We aim to reach a code coverage of > 90%. Currently, we measure the **statement** coverage at the 'check' job of the CI/CD pipeline.

Reviews

Tests are great but do not lead to better readability and maintainability. One part that will is the review process. In mango we came to the understanding that we want to review **every** change, which should be merged into the development branch. There are no exceptions to this. The idea is not only to check the code for errors, bad smells and security flaws, its part of generating a common understanding of good coding.

Linting

Another approach to improve the code quality is static code analysis, or better known as linting. Linting is an easy to set up possibility to make sure that a certain code standard is fulfilled. There are many useful rules, which can be checked automatically, so we have another line of defense and spare some time when reviewing.

Formatting

The project is formatted using black + isort with default settings.

1.11 API reference

The API reference provides detailed descriptions of mango's classes and functions.

1.11.1 Modules in mango.core

`mango.core.agent`

`mango.core.container`

`mango.core.container_protocols`

1.11.2 Modules in mango.messages

`mango.messages.codecs`

`mango.messages.message`

1.11.3 Modules in mango.modules

`mango.modules.base_module`

`mango.modules.mqtt_module`

`mango.modules.rabbit_module`

mango.modules.zero_module

1.12 Privacy Policies

We welcome you to our website. We would like to inform you about the management of your personal data in accordance with Art. 13 General Data Protection Regulation (GDPR).

Controller

The controller responsible for the described data collection and processing is OFFIS e.V., Escherweg 2, 26121 Oldenburg/Germany.

Usage Data

When you visit our website, the data collected from the use of the website is temporarily stored on our web server for statistical purposes in order to improve the quality of our website. This data set contains:

- the page, from which the data is requested
- the name of the data file,
- the date and time of the query,
- the amount of data transferred,
- the access status (file transmitted, file not found),
- a description of the type of browser used,
- the IP address of the requesting computer shortened to such an extent that no reidentification of any persona data is possible.

The listed usage data is stored anonymously. The legal basis for the processing of this personal data is provided for in Art. 6 para. 1 lit. f GDPR.

Data Transfer to Third Parties

We do not transfer your personal data to third parties.

Cookies

We use cookies on our website. Cookies are small pieces of data that are stored and read in your end-device. A distinction is made between session cookies, which are deleted when you close your browser, and permanent cookies, which are stored even after your visit has expired. Cookies may contain data that enables the recognition of the device being used. However, in some cases cookies only contain information on certain settings which are not personal data.

We use session cookies and permanent cookies on our website. The data is processed in accordance to Art. 6 para. 1 lit. f GDPR and in the interest of optimizing or enabling user guidance and improving our website presence.

Please be aware that you can set your browser to inform you when cookies are being stored or used on the website you are visiting. Thus, any use of cookies is transparent to you. You have the possibility to delete your browser configuration at any time and prevent any use of new cookies. In the event you refuse the use of cookies, please note that our web sites may not be displayed optimally and some functions are then no longer technically available.

Data Security

To avoid unauthorized access to your data, we have implemented technical and organizational measures. We use encryption technologies on our website. Your data will be transferred to our servers and back again via a connection that is protected by a TLS encryption technology. You can recognize that you are browsing on an encryption secured website by the lock-symbol shown in the address bar of your browser and by the address bar starting with https://.

Your Rights as a User

As a website user, the GDPR grants you certain rights when processing your personal data.

1. Right of access (Art. 15 GDPR): You have the right to obtain confirmation as to whether or not personal data concerning you is being processed, and, where that is the case access to the personal data and the information specified in Art. 15 GDPR.
2. Right to rectification and erasure (Art. 16 and 17 GDPR): You have the right to obtain without undue delay the rectification of inaccurate personal data concerning you and, if necessary, the right to have incomplete personal data completed. You also have the right to obtain an erasure of the personal data concerning you without undue delay, if one of the reasons listed in Art. 17 GDPR applies, e.g. if the data is no longer necessary for the intended purpose.
3. Right to restriction of processing (Art. 18 GDPR): If one of the conditions set forth in Art. 18 GDPR applies, you shall have the right to restrict the processing of your data to mere storage, e.g. if you revoke consent, to the processing, for the duration of a possible examination.
4. Right to data portability (Art. 20 GDPR): In certain situations, listed in Art. 20 GDPR, you have the right to receive the personal data concerning you in a structured, common and machine-readable format or demand a transmission of the data to another third party.
5. Right to object (Art. 21 GDPR): If the data is processed pursuant to Art. 6 para. 1 lit. f GDPR (data processing for the purposes of the legitimate interests), you have the right to object to the processing at any time for reasons arising out of your particular situation. We will then no longer process personal data, unless there are demonstrably compelling legitimate grounds for processing, which override the interests, rights and freedoms of the person concerned, or the processing serves the purpose of asserting, exercising or defending legal claims.
6. Right to lodge a complaint with a supervisory authority: Pursuant to Art. 77 GDPR, you have the right to lodge a complaint with a supervisory authority if you consider the processing of the data concerning you infringing data protection regulations. The right to lodge a complaint may be invoked in particular in the Member State of your habitual residence, place of work or the place of the alleged infringement.

Contact Details of the Data Protection Officer

Please contact our data protection officer if you have any further questions, suggestions or wishes regarding data protection:

Dr. Uwe Schläger
datenschutz nord GmbH
Web: www.datenschutz-nord-guppe.de
E-Mail: office@datenschutz-nord.de
Telefon: +49 421 69 66 32 0

1.13 Legals

Address

OFFIS e. V.
Escherweg 2
26121 Oldenburg
Germany
Phone +49 441 9722-0
Fax +49 441 9722-102
Email: institut [A T] offis.de
Internet: www.offis.de

Board Members

Prof. Dr. Sebastian Lehnhoff (Chairman)
Prof. Dr. techn. Susanne Boll-Westermann
Prof. Dr.-Ing. Andreas Hein

Register Court

Amtsgericht Oldenburg
Registernumber VR 1956

VAT Identification Number

DE 811582102

Responsible in the sense of press law

Dr. Christoph Mayer (Director)
OFFIS e.V.
Escherweg 2
26121 Oldenburg

Disclaimer

Despite careful control OFFIS assumes no liability for the content of external links. The operators of such a website are solely responsible for its content. At the time of linking the concerned sites were checked for possible violations of law. Illegal contents were not identifiable at that time. A permanent control of the linked pages is not reasonable without specific indications of a violation. Upon notification of violations, OFFIS will remove such links immediately.

1.14 Datenschutz

Datenschutzerklärung

Wir nehmen den Schutz Ihrer persönlichen Daten sehr ernst. Ihre Daten werden im Rahmen der gesetzlichen Vorschriften geschützt. Personenbezogene Daten werden auf unseren Internetseiten nur im notwendigen Umfang erhoben. In keinem Fall werden die erhobenen Daten verkauft oder aus anderen Gründen an Dritte weitergegeben.

Verantwortlicher

Verantwortlich für die hier erläuterte Datenverarbeitung ist der OFFIS e.V., Escherweg 2, 26121 Oldenburg.

Erhebung und Verarbeitung von Daten

Jeder Zugriff auf eine unserer Internetseiten und jeder Abruf einer auf den Internetseiten hinterlegten Datei werden von gängigen Webserver-Log-Dateien protokolliert. Die Speicherung dient internen systembezogenen und statistischen Zwecken. Protokolliert wird u.a.:

- welche Datei angefordert wurde,
- der Name der Datei,

- das Datum und die Uhrzeit der Anforderung,
- die übertragene Datenmenge,
- der Zugriffsstatus (Datei nicht gefunden, Datei übertragen etc.),
- der Typ des verwendeten Webbrowsers und
- die IP-Adresse des Internetseitenbesuchers

Sämtliche dieser Daten werden ausschließlich anonymisiert gespeichert und ausgewertet. Zu diesem Zweck wird die IP-Adresse des Systems, von dem aus die Internetseite oder Datei angefordert wurde, geeignet anonymisiert. Rechtsgrundlage ist Art. 6 Abs. 1 lit. f DSGVO (Datenschutz-Grundverordnung). Es ist somit weder ein Rückschluss auf eine bestimmte Person möglich, noch erfolgt eine Zusammenführung mit anderen Daten.

Cookies

Darüber hinaus verwenden wir weitere Cookies, um unsere Internetseiten besser auf Ihre Wünsche ausrichten und um statistische Daten über die Nutzung unserer Internetseiten erheben zu können. Durch diese Cookies werden keine Daten erhoben, die einen Rückschluss auf eine bestimmte Person ermöglich. Auch die Installation dieser Cookies wird durch eine entsprechende Browser-Einstellung verhindert. Einmal gesetzte Cookies können Sie jederzeit selbst löschen, indem Sie den entsprechenden Menüpunkt in Ihrem Internet-Browser aufrufen oder die Cookies auf Ihrer Festplatte löschen. Einzelheiten hierzu finden Sie im Hilfemenü Ihres Internet-Browsers. Weitergehende personenbezogene Daten werden nur erfasst, wenn Sie diese Angaben freiwillig, z.B. im Rahmen einer Anfrage oder Registrierung, machen.

Datensicherheit

Um Ihre Daten vor unerwünschten Zugriffen möglichst umfassend zu schützen, treffen wir technische und organisatorische Maßnahmen. Wir setzen auf unseren Seiten ein Verschlüsselungsverfahren ein. Ihre Angaben werden von Ihrem Rechner zu unserem Server und umgekehrt über das Internet mittels einer TLS-Verschlüsselung übertragen. Sie erkennen dies daran, dass in der Statusleiste Ihres Browsers das Schloss-Symbol geschlossen ist und die Adresszeile mit https:// beginnt. Bitte beachten Sie, dass außerhalb der vorgenannten Rahmenbedingungen, insbesondere bei der Kommunikation per E-Mail die vollständige Datensicherheit von uns naturgemäß nicht gewährleistet werden kann.

Verwendung der Daten Wir beachten den Grundsatz der zweckgebundenen Datenverwendung und erheben, verarbeiten und speichern Ihre personenbezogenen Daten nur für die Zwecke, für welche Sie uns diese mitgeteilt haben und für die technische Administration. Eine Weitergabe Ihrer persönlichen Daten an Dritte erfolgt ohne Ihre ausdrückliche Einwilligung nicht, sofern dies nicht zur Erbringung der Dienstleistung oder zur Vertragsdurchführung notwendig ist. Auch die Übermittlung an auskunftsberechtigte staatliche Institution und Behörden erfolgt nur im Rahmen der gesetzlichen Auskunftspflichten oder wenn wir durch eine gerichtliche Entscheidung zur Auskunft verpflichtet werden.

Löschung der Daten

Eine Löschung der gespeicherten personenbezogenen Daten erfolgt, wenn Sie Ihre Einwilligung zur Speicherung widerrufen, wenn deren Kenntnis zur Erfüllung des mit der Speicherung verfolgten Zwecks nicht mehr erforderlich ist oder wenn deren Speicherung aus sonstigen gesetzlichen Gründen unzulässig ist.

Ihre Rechte als Nutzer

Bei Verarbeitung Ihrer personenbezogenen Daten gewährt die DSGVO Ihnen als Webseitennutzer bestimmte Rechte:

1. Auskunftsrecht (Art. 15 DSGVO): Sie haben das Recht eine Bestätigung darüber zu verlangen, ob sie betreffende personenbezogene Daten verarbeitet werden; ist dies der Fall, so haben Sie ein Recht auf Auskunft über diese personenbezogenen Daten und auf die in Art. 15 DSGVO im einzelnen aufgeführten Informationen.
2. Recht auf Berichtigung und Löschung (Art. 16 und 17 DSGVO): Sie haben das Recht, unverzüglich die Berichtigung sie betreffender unrichtiger personenbezogener Daten und ggf. die Vervollständigung unvollständiger perso-nenbezogener Daten zu verlangen. Sie haben zudem das Recht, zu verlangen, dass sie betreffende personenbezogene Daten unverzüglich gelöscht werden, sofern einer der in Art. 17 DSGVO im einzelnen aufgeführten Gründe zutrifft, z. B. wenn die Daten für die verfolgten Zwecke nicht mehr benötigt werden.

3. Recht auf Einschränkung der Verarbeitung (Art. 18 DSGVO): Sie haben das Recht, die Einschränkung der Verarbeitung zu verlangen, wenn eine der in Art. 18 DSGVO aufgeführten Voraussetzungen gegeben ist, z. B. wenn Sie Widerspruch gegen die Verarbeitung eingelegt haben, für die Dauer einer etwai-gen Prüfung.
4. Recht auf Datenübertragbarkeit (Art. 20 DSGVO): In bestimmten Fällen, die in Art. 20 DSGVO im Einzelnen aufgeführt werden, haben Sie das Recht, die sie betreffenden personenbezogenen Daten in einem struktu-rierten, gängigen und maschinenlesbaren Format zu erhalten bzw. die Übermittlung dieser Daten an einen Dritten zu verlangen.
5. Widerspruchsrecht (Art. 21 DSGVO): Werden Daten auf Grundlage von Art. 6 Abs. 1 lit. f erhoben (Datenverarbeitung zur Wahrung berechtigter Interessen), steht Ihnen das Recht zu, aus Gründen, die sich aus Ihrer besonderen Situation ergeben, jederzeit gegen die Verarbeitung Wider-spruch einzulegen. Wir verarbeiten die personenbezogenen Daten dann nicht mehr, es sei denn, es liegen nachweisbar zwingende schutzwürdige Gründe für die Verarbeitung vor, die die Interessen, Rechte und Freiheiten der betroffenen Person überwiegen, oder die Verarbeitung dient der Geltendmachung, Ausübung oder Verteidigung von Rechtsansprüchen.
6. Beschwerderecht bei einer Aufsichtsbehörde Sie haben gem. Art. 77 DSGVO das Recht auf Beschwerde bei einer Aufsichtsbehör-de, wenn Sie der Ansicht sind, dass die Verarbeitung der Sie betreffenden Daten gegen datenschutzrechtliche Bestimmungen verstößt. Das Beschwerderecht kann insbesondere bei einer Aufsichtsbehörde in dem Mitgliedstaat Ihres Aufenthaltsorts, Ihres Arbeitsplatzes oder des Orts des mutmaßlichen Verstoßes geltend gemacht werden.

Datenschutzbeauftragter

Dr. Uwe Schläger
datenschutz nord GmbH
www.datenschutz-nord.de
office(at)datenschutz-nord.de

1.15 Impressum

Anschrift

OFFIS e. V.
Escherweg 2
26121 Oldenburg
Telefon +49 441 9722-0
Fax +49 441 9722-102
E-Mail: institut [A T] offis.de
Internet: www.offis.de

Vertretungsberechtigter Vorstand

Prof. Dr. Sebastian Lehnhoff (Vorsitzender)
Prof. Dr. techn. Susanne Boll-Westermann
Prof. Dr.-Ing. Andreas Hein

Registergericht

Amtsgericht Oldenburg
Registernummer VR 1956

Umsatzsteuer-Identifikationsnummer (USt-IdNr.)

DE 811582102

Verantwortlich im Sinne der Presse

Dr. Christoph Mayer (Bereichsleiter)
OFFIS e.V.
Escherweg 2
26121 Oldenburg

Datenschutz

Mehr zum Thema Datenschutz finden Sie *hier*.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search