SPADE Documentation

Release 3.2.2

Javi Palanca

Aug 01, 2022

Contents

Contents:

SPADE

Smart Python Agent Development Environment

A multi-agent systems platform written in Python and based on instant messaging (XMPP).

Develop agents that can chat both with other agents and humans.

- Free software: MIT license

- Documentation: http://spade-mas.readthedocs.io/

1.1 Features

- Multi-agent platform based on XMPP

- Presence notification allows the system to know the current state of the agents in real-time

- Python >=3.6

- Asyncio-based

- Agent model based on behaviours

- Supports FIPA metadata using XMPP Data Forms (XEP-0004: Data Forms)

- Web-based interface
- Use any XMPP server

1.2 Plugins

- **spade_bdi (BDI agents with AgentSpeak):**

    - Code: https://github.com/javipalanca/spade_bdi

- **spade_pubsub (PubSub protocol for agents):**

    - Code: https://github.com/javipalanca/spade_pubsub

    - documentation: https://spade-pubsub.readthedocs.io

- **spade_artifact (Artifacts for SPADE):**

    - Code: https://github.com/javipalanca/spade_artifact

    - Documentation: https://spade-artifact.readthedocs.io

- **spade_bokeh (bokeh plots for agents):**

    - Code: https://github.com/javipalanca/spade_bokeh

    - Documentation: https://spade-bokeh.readthedocs.io

1.3 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

Foreword

The idea of an XMPP-based agent platform appeared one night at 4 A.M. when, studying the features of the Jabber architecture, we found out great similarities with the ones of a FIPA-compliant agent platform. The XMPP protocol offered a great architecture for agents to communicate in a structured way and solved many issues present when designing a platform, such as authenticating the users (the agents), provide a directory or create communication channels.

We started to work on our first prototype of this Jabber-powered platform and within a week we had a small working proof of concept by the name of *Fipper* which eventually allowed for dumb agents to connect and communicate through a common XMPP server.

Since that day, things have changed a bit. The small proof of concept evolved into a full-featured FIPA platform, and the new SPADE name was coined. As usual, we later had to find the meaning of the beautiful acronym. We came up with **Smart Python multi-Agent Development Environment**, which sounded both good and geek enough.

The years passed, and everything evolved except the platform. Python reached version 3, which came with lots of interesting changes and improvements. We also became better programmers (just because of the grounding and the experience that the years give), we met the PEP8 and the Clean Code principles and they opened our eyes to a new world. That's why in 2017 SPADE was fully rewritten in Python 3.6, using asyncio and strictly following PEP8 and Clean Code principles.

We hope you like this software and have as much fun using it as we had writing it. Of course we also hope that it may become useful, but that is a secondary matter.

The SPADE agent model

The Agent Model is basically composed of a connection mechanism to the platform, a message dispatcher, and a set of different behaviours that the dispatcher gives the messages to. Every agent needs an identifier called Jabber ID a.k.a. JID and a valid password to establish a connection with the XMPP server.

The JID (composed by a username, an @, and a server domain) will be the name that identifies an agent in the platform, e.g. *myagent@myprovider.com*.

3.1 Connection to the platform

Communications in SPADE are handled internally by means of the XMPP protocol. This protocol has a mechanism to register and authenticate users against an XMPP server.

After a succesful register, each agent holds an open and persistent XMPP stream of communications with the platform. This process is automatically triggered as part of the agent registration process.

3.2 The message dispatcher

Each SPADE agent has an internal message dispatcher component. This message dispatcher acts as a mailman: when a message for the agent arrives, it places it in the correct "mailbox" (more about that later); and when the agent needs to send a message, the message dispatcher does the job, putting it in the communication stream. The message dispatching is done automatically by the SPADE agent library whenever a new message arrives or is to be sent.

3.3 The behaviours

An agent can run serveral behaviours simultaneously. A behaviour is a task that an agent can execute using repeating patterns. SPADE provides some predefined behaviour types: Cyclic, One-Shot, Periodic, Time-Out and Finite State Machine. Those behaviour types help to implement the different tasks that an agent can perform. The kind of behaviours supported by a SPADE agent are the following:

- Cyclic and Periodic behaviours are useful for performing repetitive tasks.

- One-Shot and Time-Out behaviours can be used to perform casual tasks.

- The Finite State Machine allows more complex behaviours to be built.

Every agent can have as many behaviours as desired. When a message arrives to the agent, the message dispatcher redirects it to the correct behaviour queue. A behaviour has a message template attached to it. Therefore, the message dispatcher uses this template to determine which behaviour the message is for, by matching it with the correct template. A behaviour can thus select what kind of messages it wants to receive by using templates.

CHAPTER 4

Installation

4.1 Stable release

To install SPADE, run this command in your terminal:

```
$ pip install spade
```

This is the preferred method to install SPADE, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

4.2 From sources

The sources for SPADE can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/javipalanca/spade
```

Or download the tarball:

```
$ curl  -OL https://github.com/javipalanca/spade/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

Quick Start

5.1 Creating your first dummy agent

It's time for us to build our first SPADE agent. We'll assume that we have a registered user in an XMPP server with a jid and a password. The jid contains the agent's name (before the @) and the DNS or IP of the XMPP server (after the @). But **remember**! You should have your own jid and password in an XMPP server running in your own computer or in the Internet. In this example we will assume that our jid is *your_jid@your_xmpp_server* and the password is *your_password*.

Hint: To create a new XMPP account you can follow the steps of https://xmpp.org/getting-started/

Hint: To install an XMPP server visit https://xmpp.org/software/servers.html (we recommend Prosody IM)

A basic SPADE agent is really a python script that imports the spade module and that uses the constructs defined therein. For starters, fire up you favorite Python editor and create a file called dummyagent.py.

> **Warning:** Remember to change the example's jids and passwords by your own accounts. These accounts do not exist and are only for demonstration purposes.

To create an agent in a project you just need to:

```python
from spade import agent, quit_spade

class DummyAgent(agent.Agent):
    async def setup(self):
        print("Hello World! I'm agent {}".format(str(self.jid)))

dummy = DummyAgent("your_jid@your_xmpp_server", "your_password")
```

```
future = dummy.start()
future.result()

dummy.stop()
quit_spade()
```

This agent is only printing on screen a message during its setup and stopping. If you run this script you get the following output:

```
$ python dummyagent.py
Hello World! I'm agent your_jid@your_xmpp_server
$
```

And that's it! We have built our first SPADE Agent in 6 lines of code. Easy, isn't it? Of course, this is a very very dumb agent that does nothing, but it serves well as a starting point to understand the logics behind SPADE.

Note: Note how the `start` function returns a future (or promise) which you can wait for with the result method (`future.result()`) to make sure that the `start` coroutine has finished before invoking the `stop` coroutine.

5.2 An agent with a behaviour

Let's build a more functional agent, one that uses an actual behaviour to perform a task. As we stated earlier, the real programming of the SPADE agents is done mostly in the behaviours. Let's see how.

Let's create a cyclic behaviour that performs a task. In this case, a simple counter. We can modify our existing `dummyagent.py` script.

Warning: Remember to change the example's jids and passwords by your own accounts. These accounts do not exist and are only for demonstration purposes.

Example:

```python
import time
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour

class DummyAgent(Agent):
    class MyBehav(CyclicBehaviour):
        async def on_start(self):
            print("Starting behaviour . . .")
            self.counter = 0

        async def run(self):
            print("Counter: {}".format(self.counter))
            self.counter += 1
            await asyncio.sleep(1)

    async def setup(self):
        print("Agent starting . . .")
```

```python
        b = self.MyBehav()
        self.add_behaviour(b)

if __name__ == "__main__":
    dummy = DummyAgent("your_jid@your_xmpp_server", "your_password")
    future = dummy.start()
    future.result()

    print("Wait until user interrupts with ctrl+C")
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print("Stopping...")
    dummy.stop()
```

As you can see, we have defined a custom behaviour called MyBehav that inherits from the spade.behaviour.CyclicBehaviour class, the default class for all behaviours. This class represents a cyclic behaviour with no specific period, that is, a loop-like behaviour.

You can see that there is a coroutine called `on_start()` in the behaviour. This method is similar to the `setup()` method of the agent class but it is run in the async loop. It is executed just before the main iteration of the behaviour begins and it is used for initialization code. In this case, we print a line and initialize the variable for the counter. There is also an `on_end()` coroutine that is executed when a behaviour is done or killed.

Also, there is the `run()` method, which is very important. In all behaviours, this is the method in which the core of the programming is done, because this method is called on each iteration of the behaviour loop. It acts as the body of the loop, sort of. In our example, the `run()` method prints the current value of the counter, increases it and then waits for a second (to iterate again).

> **Warning:** **Note** that the `run()` method is an async coroutine!. This is very important since SPADE is an **async library** based on python's asyncio. That's why we can call async methods inside the `run()` method, like the `await asyncio.sleep(1)`, which sleeps during one second without blocking the event loop.

Now look at the `setup()` coroutine of the agent. There, we make an instance of MyBehav and add it to the current agent by means of the `add_behaviour()` method. The first parameter of this method is the behaviour we want to add, and there is also a second optional parameter which is the template associated to that behaviour, but we will talk later about templates.

Let's test our new agent:

```
$ python dummyagent.py
Agent starting . . .
Starting behaviour . . .
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
```

. . . and so on. As we have not set any end condition, this agent would go on counting forever until we press ctrl+C.

5.3 Finishing a behaviour

If you want to finish a behaviour you can kill it by using the `self.kill(exit_code)` method. This method **marks** the behaviour to be killed at the next loop iteration and stores the exit_code to be queried later.

An example of how to kill a behaviour:

```python
import time
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour

class DummyAgent(Agent):
    class MyBehav(CyclicBehaviour):
        async def on_start(self):
            print("Starting behaviour . . .")
            self.counter = 0

        async def run(self):
            print("Counter: {}".format(self.counter))
            self.counter += 1
            if self.counter > 3:
                self.kill(exit_code=10)
                return
            await asyncio.sleep(1)

        async def on_end(self):
            print("Behaviour finished with exit code {}.".format(self.exit_code))

    async def setup(self):
        print("Agent starting . . .")
        self.my_behav = self.MyBehav()
        self.add_behaviour(self.my_behav)

if __name__ == "__main__":
    dummy = DummyAgent("your_jid@your_xmpp_server", "your_password")
    future = dummy.start()
    future.result()  # Wait until the start method is finished

    # wait until user interrupts with ctrl+C
    while not dummy.my_behav.is_killed():
        try:
            time.sleep(1)
        except KeyboardInterrupt:
            break
    dummy.stop()
```

And the output of this example would be:

```
$ python killbehav.py
Agent starting . . .
Starting behaviour . . .
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Behaviour finished with exit code 10.
```

Note: An exit code may be of any type you need: int, dict, string, exception, etc.

> **Warning:** Remember that killing a behaviour does not cancel its current run loop, if you need to finish the current iteration you'll have to call return.

Hint: If a exception occurs inside an `on_start`, `run` or `on_end` coroutines, the behaviour will be automatically killed and the exception will be stored as its `exit_code`.

5.4 Finishing SPADE

There is a helper to quickly finish all the agents and behaviors running in your process. This helper function is `quit_spade`:

```python
from spade import quit_spade

from spade import agent

class DummyAgent(agent.Agent):
    async def setup(self):
        print("Hello World! I'm agent {}".format(str(self.jid)))

dummy = DummyAgent("your_jid@your_xmpp_server", "your_password")
future = dummy.start()
future.result()

dummy.stop()

quit_spade()
```

Hint: The `quit_spade` helper is not mandatory, but it helps to terminate all agents of the active container along with their behaviors, as well as free all pending resources (threads, etc...).

5.5 Creating an agent from within another agent

There is a common use case where you may need to create an agent from within another agent, that is, from within another agent's behaviour. This is a *special* case because you can't create a new event loop when you have a loop already running. For this special case you can use the `start` method as usual. But in this case `` `start `` behaves as a coroutine, so it MUST be called with an `await` statement in order to work properly. Example:

```python
from spade import quit_spade
from spade.agent import Agent
from spade.behaviour import OneShotBehaviour


class AgentExample(Agent):
```

```python
    async def setup(self):
        print(f"{self.jid} created.")


class CreateBehav(OneShotBehaviour):
    async def run(self):
        agent2 = AgentExample("agent2_example@your_xmpp_server", "fake_password")
        # This start is inside an async def, so it must be awaited
        await agent2.start(auto_register=True)


if __name__ == "__main__":
    agent1 = AgentExample("agent1_example@your_xmpp_server", "fake_password")
    behav = CreateBehav()
    agent1.add_behaviour(behav)
    # This start is in a synchronous piece of code, so it must NOT be awaited
    future = agent1.start(auto_register=True)
    future.result()

    # wait until the behaviour is finished to quit spade.
    behav.join()
    quit_spade()
```

Warning: Remember to call `start` with an `await` whenever you are inside an asyncronous method (another coroutine). Otherwise, call `start` as usual (without the `await` statement).

Note: The `stop` method behaves just like `start`. They change depending on the context. They return a coroutine or a future depending on whether they are called from a coroutine or a synchronous method.

Agent communications

6.1 Using templates

Templates is the method used by SPADE to dispatch received messages to the behaviour that is waiting for that message. When adding a behaviour you can set a template for that behaviour, which allows the agent to deliver a message received by the agent to that registered behaviour. A `Template` instance has the same attributes of a `Message` and all the attributes defined in the template must be equal in the message for this to match.

The attributes that can be set in a template are:

- **to**: the jid string of the receiver of the message.

- **sender** the jid string of the sender of the message.

- **body**: the body of the message.

- **thread**: the thread id of the conversation.

- **metadata**: a (key, value) dictionary of strings to define metadata of the message. This is useful, for example, to include FIPA attributes like *ontology*, *performative*, *language*, etc.

An example of template matching:

```
template = Template()
template.sender = "sender1@host"
template.to = "recv1@host"
template.body = "Hello World"
template.thread = "thread-id"
template.metadata = {"performative": "query"}

message = Message()
message.sender = "sender1@host"
message.to = "recv1@host"
message.body = "Hello World"
message.thread = "thread-id"
message.set_metadata("performative", "query")
```

```
assert template.match(message)
```

Templates also support boolean operators to create more complex templates. Bitwise operators (&, |, ^ and ~) may be used to combine simpler templates.

- **&**: Does a boolean AND between templates.

- **|**: Does a boolean OR between templates.

- **^**: Does a boolean XOR between templates.

- **~**: Returns the complement of the template.

Some examples of these operators:

```
t1 = Template()
t1.sender = "sender1@host"
t2 = Template()
t2.to = "recv1@host"
t2.metadata = {"performative": "query"}


m = Message()
m.sender = "sender1@host"
m.to = "recv1@host"
m.metadata = {"performative": "query"}

# And AND operator
assert (t1 & t2).match(m)

t3 = Template()
t3.sender = "not_valid_sender@host"

# A NOT complement operator
assert (~t3).match(m)
```

6.2 Sending and Receiving Messages

As you know, messages are the basis of every MAS. They are the centre of the whole "computing as interaction" paradigm in which MAS are based. So it is very important to understand which facilities are present in SPADE to work with agent messages.

First and foremost, threre is a `Message` class. This class is `spade.message.Message` and you can instantiate it to create new messages to work with. The class provides a method to introduce metadata into messages, this is useful for using the fields present in standard FIPA-ACL Messages. When a message is ready to be sent, it can be passed on to the send() method of the behaviour, which will trigger the internal communication process to actually send it to its destination. Note that the send function is an async coroutine, so it needs to be called with an `await` statement.

> **Warning:** Remember to change the example's jids and passwords by your own accounts. These accounts do not exist and are only for demonstration purposes.

Here is a self-explaining example:

```python
import time
from spade.agent import Agent
from spade.behaviour import OneShotBehaviour
from spade.message import Message


class SenderAgent(Agent):
    class InformBehav(OneShotBehaviour):
        async def run(self):
            print("InformBehav running")
            msg = Message(to="receiver@your_xmpp_server")     # Instantiate the
→message
            msg.set_metadata("performative", "inform")  # Set the "inform" FIPA
→performative
            msg.set_metadata("ontology", "myOntology")  # Set the ontology of the
→message content
            msg.set_metadata("language", "OWL-S")       # Set the language of the
→message content
            msg.body = "Hello World"                    # Set the message content

            await self.send(msg)
            print("Message sent!")

            # set exit_code for the behaviour
            self.exit_code = "Job Finished!"

            # stop agent from behaviour
            await self.agent.stop()

    async def setup(self):
        print("SenderAgent started")
        self.b = self.InformBehav()
        self.add_behaviour(self.b)


if __name__ == "__main__":
    agent = SenderAgent("sender@your_xmpp_server", "sender_password")
    future = agent.start()
    future.result()

    while agent.is_alive():
        try:
            time.sleep(1)
        except KeyboardInterrupt:
            agent.stop()
            break
    print("Agent finished with exit code: {}".format(agent.b.exit_code))
```

This code would output:

```
$ python sender.py
SenderAgent started
InformBehav running
Message sent!
Agent finished with exit code: Job Finished!
```

Ok, we have sent a message but now we need someone to receive that message. Show me the code:

```python
import time
from spade.agent import Agent
from spade.behaviour import OneShotBehaviour
from spade.message import Message
from spade.template import Template


class SenderAgent(Agent):
    class InformBehav(OneShotBehaviour):
        async def run(self):
            print("InformBehav running")
            msg = Message(to="receiver@your_xmpp_server")     # Instantiate the
→message
            msg.set_metadata("performative", "inform")  # Set the "inform" FIPA
→performative
            msg.body = "Hello World"                    # Set the message content

            await self.send(msg)
            print("Message sent!")

            # stop agent from behaviour
            await self.agent.stop()

    async def setup(self):
        print("SenderAgent started")
        b = self.InformBehav()
        self.add_behaviour(b)

class ReceiverAgent(Agent):
    class RecvBehav(OneShotBehaviour):
        async def run(self):
            print("RecvBehav running")

            msg = await self.receive(timeout=10) # wait for a message for 10 seconds
            if msg:
                print("Message received with content: {}".format(msg.body))
            else:
                print("Did not received any message after 10 seconds")

            # stop agent from behaviour
            await self.agent.stop()

    async def setup(self):
        print("ReceiverAgent started")
        b = self.RecvBehav()
        template = Template()
        template.set_metadata("performative", "inform")
        self.add_behaviour(b, template)


if __name__ == "__main__":
    receiveragent = ReceiverAgent("receiver@your_xmpp_server", "receiver_password")
    future = receiveragent.start()
    future.wait() # wait for receiver agent to be prepared.
    senderagent = SenderAgent("sender@your_xmpp_server", "sender_password")
    senderagent.start()
```

```python
    while receiveragent.is_alive():
        try:
            time.sleep(1)
        except KeyboardInterrupt:
            senderagent.stop()
            receiveragent.stop()
            break
    print("Agents finished")
```

Note: It's important to remember that the send and receive functions are **coroutines**, so they **always** must be called with the `await` statement.

In this example you can see how the `RecvBehav` behaviour receives the message because the template includes a *performative* with the value **inform** in the metadata and the sent message does also include that metadata, so the message and the template match.

You can also note that we are using an *ugly* `time.sleep` to introduce an explicit wait to avoid sending the message before the receiver agent is up and ready since in another case the message would never be received (remember that spade is a **real-time** messaging platform. In future sections we'll show you how to use *presence notification* to wait for an agent to be *available*.

The code below would output:

```
$ python send_and_recv.py
ReceiverAgent started
RecvBehav running
SenderAgent started
InformBehav running
Message sent!
Message received with content: Hello World
Agents finished
```

Advanced Behaviours

There are more complex types of behaviours that you can use in SPADE. Let's see some of them.

7.1 Periodic Behaviour

This behaviour runs its `run()` body at a scheduled `period`. This period is set in seconds. You can also delay the startup of the periodic behaviour by setting a datetime in the `start_at` parameter.

> **Warning:** Remember to change the example's jids and passwords by your own accounts. These accounts do not exist and are only for demonstration purposes.

Let's see an example:

```python
import datetime
import getpass
import time

from spade import quit_spade
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour, PeriodicBehaviour
from spade.message import Message


class PeriodicSenderAgent(Agent):
    class InformBehav(PeriodicBehaviour):
        async def run(self):
            print(f"PeriodicSenderBehaviour running at {datetime.datetime.now().
→time()}: {self.counter}")
            msg = Message(to=self.get("receiver_jid"))  # Instantiate the message
            msg.body = "Hello World"  # Set the message content
```

(continues on next page)

```python
            await self.send(msg)
            print("Message sent!")

            if self.counter == 5:
                self.kill()
            self.counter += 1

        async def on_end(self):
            # stop agent from behaviour
            await self.agent.stop()

        async def on_start(self):
            self.counter = 0

    async def setup(self):
        print(f"PeriodicSenderAgent started at {datetime.datetime.now().time()}")
        start_at = datetime.datetime.now() + datetime.timedelta(seconds=5)
        b = self.InformBehav(period=2, start_at=start_at)
        self.add_behaviour(b)


class ReceiverAgent(Agent):
    class RecvBehav(CyclicBehaviour):
        async def run(self):
            print("RecvBehav running")
            msg = await self.receive(timeout=10)  # wait for a message for 10 seconds
            if msg:
                print("Message received with content: {}".format(msg.body))
            else:
                print("Did not received any message after 10 seconds")
                self.kill()

        async def on_end(self):
            await self.agent.stop()

    async def setup(self):
        print("ReceiverAgent started")
        b = self.RecvBehav()
        self.add_behaviour(b)


if __name__ == "__main__":
    receiver_jid = input("Receiver JID> ")
    passwd = getpass.getpass()
    receiveragent = ReceiverAgent(receiver_jid, passwd)

    sender_jid = input("Sender JID> ")
    passwd = getpass.getpass()
    senderagent = PeriodicSenderAgent(sender_jid, passwd)

    future = receiveragent.start(auto_register=True)
    future.result()  # wait for receiver agent to be prepared.

    senderagent.set("receiver_jid", receiver_jid)  # store receiver_jid in the sender␣
↪knowledge base
    senderagent.start(auto_register=True)
```

```python
    while receiveragent.is_alive():
        try:
            time.sleep(1)
        except KeyboardInterrupt:
            senderagent.stop()
            receiveragent.stop()
            break
    print("Agents finished")
    quit_spade()
```

The output of this code would be similar to:

```
$ python periodic.py
ReceiverAgent started
RecvBehav running
PeriodicSenderAgent started at 17:40:39.901903
PeriodicSenderBehaviour running at 17:40:45.720227: 0
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:46.906229: 1
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:48.906347: 2
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:50.903576: 3
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:52.905082: 4
Message sent!
Message received with content: Hello World
RecvBehav running
PeriodicSenderBehaviour running at 17:40:54.904886: 5
Message sent!
Message received with content: Hello World
RecvBehav running
Did not received any message after 10 seconds
Agents finished
```

7.2 TimeoutBehaviour

You can also create a `TimeoutBehaviour` which is run once (like OneShotBehaviours) but its activation is triggered at a specified `datetime` just as in `PeriodicBehaviours`.

Let's see an example:

```python
import getpass
import time
import datetime
from spade.agent import Agent
```

```python
from spade.behaviour import CyclicBehaviour, TimeoutBehaviour
from spade.message import Message


class TimeoutSenderAgent(Agent):
    class InformBehav(TimeoutBehaviour):
        async def run(self):
            print(f"TimeoutSenderBehaviour running at {datetime.datetime.now().time()}
→")
            msg = Message(to=self.get("receiver_jid"))  # Instantiate the message
            msg.body = "Hello World"  # Set the message content

            await self.send(msg)

        async def on_end(self):
            await self.agent.stop()

    async def setup(self):
        print(f"TimeoutSenderAgent started at {datetime.datetime.now().time()}")
        start_at = datetime.datetime.now() + datetime.timedelta(seconds=5)
        b = self.InformBehav(start_at=start_at)
        self.add_behaviour(b)


class ReceiverAgent(Agent):
    class RecvBehav(CyclicBehaviour):
        async def run(self):
            msg = await self.receive(timeout=10)  # wait for a message for 10 seconds
            if msg:
                print("Message received with content: {}".format(msg.body))
            else:
                print("Did not received any message after 10 seconds")
                self.kill()

        async def on_end(self):
            await self.agent.stop()

    async def setup(self):
        b = self.RecvBehav()
        self.add_behaviour(b)


if __name__ == "__main__":
    receiver_jid = input("Receiver JID> ")
    passwd = getpass.getpass()
    receiveragent = ReceiverAgent(receiver_jid, passwd)

    sender_jid = input("Sender JID> ")
    passwd = getpass.getpass()
    senderagent = TimeoutSenderAgent(sender_jid, passwd)

    future = receiveragent.start(auto_register=True)
    future.result()  # wait for receiver agent to be prepared.

    senderagent.set("receiver_jid", receiver_jid)  # store receiver_jid in the sender
→knowledge base
    senderagent.start(auto_register=True)
```

```python
    while receiveragent.is_alive():
        try:
            time.sleep(1)
        except KeyboardInterrupt:
            senderagent.stop()
            receiveragent.stop()
            break
    print("Agents finished")
```

This would produce the following output:

```
$python timeout.py
TimeoutSenderAgent started at 18:12:09.620316
TimeoutSenderBehaviour running at 18:12:14.625403
Message received with content: Hello World
Did not received any message after 10 seconds
Agents finished
```

7.3 Finite State Machine Behaviour

SPADE agents can also have more complex behaviours which are a finite state machine (FSM) which has registered states and transitions between states. This kind of behaviour allows SPADE agents to build much more complex and interesting behaviours in our agent model.

The `FSMBehaviour` class is a container behaviour (subclass of `CyclicBehaviour`) that implements the methods `add_state(name, state, initial)` and `add_transition(source, dest)`. Every state of the FSM must be registered in the behaviour with a string name and an instance of the `State` class. This `State` class represents a node of the FSM and (since it's a subclass of `OneShotBehaviour`) you must override the `run` coroutine just as in a regular behaviour. Since a `State` is a regular behaviour, you can also override the `on_start` and `on_end` coroutines, and, of course, use the `send` and `receive` coroutines to be able to interact with other agents via SPADE messaging.

Note: To mark a `State` as initial state of the FSM set **initial** parameter to *True* when calling *add_state* (`add_state(name, state, initial=True)`). **A FSM can only have ONE initial state, so the initial state will be the last one registered.**

Transitions in a `FSMBehaviour` define from which state to which state it is allowed to transit. A `State` defines its transit to another state by using the `set_next_state` method in its `run` coroutine. By using the `set_next_state` method a state dinamically expresses to which state it transits when it finishes. After running a state, the FSM reads this *next_state* value and, if the transition is valid, it transits to that state.

Warning: If the transition is not registered it raises a `NotValidTransition` exception and the FSM behaviour is finished.

Warning: `set_next_state` must be called with the same string name with which that state was registered. If the state is not registered a `NotValidState` exception is raised and the FSM behaviour is finished.

A `FSMBehaviour` has a unique template, which is shared with all the states of the FSM. You must take this into account when you describe your FSM states, because they will share the same message queue.

Next, we are going to see an example where a very simple FSM is defined, with three states, which transitate from one state to the next one in order. It also sends a message to itself at the first initial state, which is received at the third (and final) state. Also note that the third state is a final state because it does not set a *next_state* to transit to:

```python
import time

from spade.agent import Agent
from spade.behaviour import FSMBehaviour, State
from spade.message import Message

STATE_ONE = "STATE_ONE"
STATE_TWO = "STATE_TWO"
STATE_THREE = "STATE_THREE"


class ExampleFSMBehaviour(FSMBehaviour):
    async def on_start(self):
        print(f"FSM starting at initial state {self.current_state}")

    async def on_end(self):
        print(f"FSM finished at state {self.current_state}")
        await self.agent.stop()


class StateOne(State):
    async def run(self):
        print("I'm at state one (initial state)")
        msg = Message(to=str(self.agent.jid))
        msg.body = "msg_from_state_one_to_state_three"
        await self.send(msg)
        self.set_next_state(STATE_TWO)


class StateTwo(State):
    async def run(self):
        print("I'm at state two")
        self.set_next_state(STATE_THREE)


class StateThree(State):
    async def run(self):
        print("I'm at state three (final state)")
        msg = await self.receive(timeout=5)
        print(f"State Three received message {msg.body}")
        # no final state is setted, since this is a final state


class FSMAgent(Agent):
    async def setup(self):
        fsm = ExampleFSMBehaviour()
        fsm.add_state(name=STATE_ONE, state=StateOne(), initial=True)
        fsm.add_state(name=STATE_TWO, state=StateTwo())
        fsm.add_state(name=STATE_THREE, state=StateThree())
        fsm.add_transition(source=STATE_ONE, dest=STATE_TWO)
        fsm.add_transition(source=STATE_TWO, dest=STATE_THREE)
```

```python
        self.add_behaviour(fsm)


if __name__ == "__main__":
    fsmagent = FSMAgent("fsmagent@your_xmpp_server", "your_password")
    future = fsmagent.start()
    future.result()

    while fsmagent.is_alive():
        try:
            time.sleep(1)
        except KeyboardInterrupt:
            fsmagent.stop()
            break
    print("Agent finished")
```

7.4 Waiting a Behaviour

Sometimes you may need to wait for a behaviour to finish. In order to make this easy, behaviours provide a method called `join`. Using this method you can wait for a behaviour to be finished. Be careful, since this is a blocking operation. In order to make it usable inside and outside coroutines, this is also a morphing method (like `start` and `stop`) which behaves different depending on the context. It returns a coroutine or a future depending on whether it is called from a coroutine or a synchronous method. Example:

```python
import asyncio
import getpass

from spade import quit_spade
from spade.agent import Agent
from spade.behaviour import OneShotBehaviour


class DummyAgent(Agent):
    class LongBehav(OneShotBehaviour):
        async def run(self):
            await asyncio.sleep(5)
            print("Long Behaviour has finished")

    class WaitingBehav(OneShotBehaviour):
        async def run(self):
            await self.agent.behav.join()  # this join must be awaited
            print("Waiting Behaviour has finished")

    async def setup(self):
        print("Agent starting . . .")
        self.behav = self.LongBehav()
        self.add_behaviour(self.behav)
        self.behav2 = self.WaitingBehav()
        self.add_behaviour(self.behav2)


if __name__ == "__main__":

    jid = input("JID> ")
```

```
    passwd = getpass.getpass()

    dummy = DummyAgent(jid, passwd)
    future = dummy.start()
    future.result()

    dummy.behav2.join()  # this join must not be awaited

    print("Stopping agent.")
    dummy.stop()

    quit_spade()
```

Presence Notification

One of the most differentiating features of SPADE agents is their ability to maintain a roster or list of contacts (friends) and to receive notifications in real time about their contacts. This is a feature inherited from instant messaging technology and that, thanks to XMPP, SPADE powers to the maximum for its agents.

8.1 Presence Manager

Every SPADE agent has a property to manage its presence. This manager is called `presence` and implements all the methods and attributes to manage an agent's presence notification.

A presence object has three attributes: the **state**, the **status** and the **priority**. Let's see every one of them:

8.1.1 State

The state of a presence message shows if the agent is **Available** or **Unavailable**. This means that the agent is connected to an XMPP server or not. This is very useful to know, before contacting an agent, if it is available to receive a message in real time or not. The availability state is a boolean attribute.

Besides, the *State* has also an attribute to give additional information about *how available* the contact is. This is the **Show** attribute. The *Show* attribute belongs to the class `aioxmpp.PresenceShow` and can take the following values:

- `PresenceShow.CHAT`: The entity or resource is actively interested in chatting (i.e. receiving messages).

- `PresenceShow.AWAY`: The entity or resource is temporarily away, however it can receive messages (they will probably be attended later)

- `PresenceShow.XA`: The entity or resource is away for an extended period (xa = "*eXtended Away*").

- `PresenceShow.DND`: The entity or resource is busy (dnd = "*Do Not Disturb*").

- `PresenceShow.NONE`: Signifies absence of the *Show* element. Used for unavailable states.

An agent can set its availability and show property:

```
agent.presence.set_available(availability=True, show=PresenceShow.CHAT)
```

> **Warning:** If you set your presence to *unavailable* the only possible show state is `PresenceShow.NONE`.

A short method to set *unavailability* is:

```
agent.presence.set_unavailable()
```

To get your presence state:

```
my_state = agent.presence.state  # Gets your current PresenceState instance.

agent.presence.is_available()  # Returns a boolean to report wether the agent is
→available or not

my_show = agent.presence.state.show  # Gets your current PresenceShow info.
```

> **Tip:** If no *Show* element is provided, the entity is assumed to be online and available.

8.1.2 Status

The status is used to set a textual status to your presence information. It is used to explain with natural language your current status which is broadcasted when the client connects and when the presence is re-emitted.

An agent can get its status as follows:

```
>> agent.presence.status
{None: "Working..."}
```

> **Warning:** It should be noted that the status is returned as a dict with a `None` key. This is because the status supports different languages. If you set the status as a string it is set as the default status (and stored with the key `None`. If you want to set the status in different languages you can specify it using the keys:
>
> ```
> >> agent.presence.status
> {
>   None: "Working...",
>   "es": "Trabajando...",
>   "fr": "Travailler..."
> }
> ```

8.1.3 Priority

Since an agent (and indeed any XMPP user) can have multiple connections to an XMPP server, it can set the priority of each of those connections to establish the level of each one. The value must be an integer between -128 and +127.

8.1.4 Setting the Presence

There is a method that can be used to set the three presence attributes. Since they are all optional, you can change any of the attribute values with every call:

```
agent.presence.set_presence(
                            state=PresenceState(True, PresenceShow.CHAT),  #␣
→available and interested in chatting
                            status="Lunch",
                            priority=2
                            )
```

8.2 Availability handlers

To get notified when a contact gets available or unavailable you can override the `on_available` and `on_unavailable` handlers. As you can see in the next example, these handlers receive the peer jid of the contact and the *stanza* of the XMPP Presence message (class `aioxmpp.Presence`) which contains all its presence information (availability, show, state, priority, . . .):

```
def my_on_available_handler(peer_jid, stanza):
    print(f"My friend {peer_jid} is now available with show {stanza.show}")

agent.presence.on_available =  my_on_available_handler
```

8.3 Contact List

Every contact to whom you are subscribed to appears in your *contact list*. You can use the `get_contacts()` method to get the full list of your contacts. This method returns a `dict` where the keys are the `JID` of your contacts and the values are an dict that show the information you have about each of your contacts (presence, name, approved, groups, ask, subscription, . . .). Note that the "presence" value is an `aioxmpp.Presence` object with the latest updated information about the contact's presence.

Example:

```
>>> contacts = agent.presence.get_contacts()
>>> contacts[myfriend_jid]
    {
      'presence': Presence(type_=PresenceType.AVAILABLE),
      'subscription': 'both',
      'name': 'My Friend',
      'approved': True
    }
```

> **Warning:** An empty contact list will return an empty dictionary.

8.4 Subscribing and unsubscribing to contacts

To subscribe and unsubscribe to/from a contact you have to send a special presence message asking for that subscription. SPADE helps you by providing some methods that send these special messages:

```
# Send a subscription request to a peer_jid
agent.presence.subscribe(peer_jid)

# Send an unsubscribe request to a peer_jid
agent.presence.unsubscribe(peer_jid)
```

8.4.1 Subscription handlers

The way you have to get notified when someone wants to subscribe/unsubscribe to you or when you want to get notified if a subscription/unsubscription process has succeed is by means of handlers. There are four handlers that you can override to manage these kind of messages: `on_subscribe`, `on_unsubscribe`, `on_subscribed` and `on_unsubscribed`:

```python
def my_on_subscribe_callback(peer_jid):
    if i_want_to_approve_request:
        self.approve(peer_jid)

agent.presence.on_subscribe = my_on_subscribe_callback
```

Note: In the previous example you can see also how to approve a subscription request by using the `approve` method.

Tip: If you want to automatically approve all subscription requests you can set the `approve_all` flag to `True`.

8.5 Example

This is an example that shows in a practical way the presence module:

```python
import time
import getpass

from spade.agent import Agent
from spade.behaviour import OneShotBehaviour


class Agent1(Agent):
    async def setup(self):
        print("Agent {} running".format(self.name))
        self.add_behaviour(self.Behav1())

    class Behav1(OneShotBehaviour):
        def on_available(self, jid, stanza):
            print("[{}] Agent {} is available.".format(self.agent.name, jid.split("@
→")[0]))

        def on_subscribed(self, jid):
            print("[{}] Agent {} has accepted the subscription.".format(self.agent.
→name, jid.split("@")[0]))
            print("[{}] Contacts List: {}".format(self.agent.name, self.agent.
→presence.get_contacts()))
```

(continues on next page)

```python
    def on_subscribe(self, jid):
        print("[{}] Agent {} asked for subscription. Let's aprove it.".
→format(self.agent.name, jid.split("@")[0]))
        self.presence.approve(jid)

    async def run(self):
        self.presence.on_subscribe = self.on_subscribe
        self.presence.on_subscribed = self.on_subscribed
        self.presence.on_available = self.on_available

        self.presence.set_available()
        self.presence.subscribe(self.agent.jid2)


class Agent2(Agent):
    async def setup(self):
        print("Agent {} running".format(self.name))
        self.add_behaviour(self.Behav2())

    class Behav2(OneShotBehaviour):
        def on_available(self, jid, stanza):
            print("[{}] Agent {} is available.".format(self.agent.name, jid.split("@
→")[0]))

        def on_subscribed(self, jid):
            print("[{}] Agent {} has accepted the subscription.".format(self.agent.
→name, jid.split("@")[0]))
            print("[{}] Contacts List: {}".format(self.agent.name, self.agent.
→presence.get_contacts()))

        def on_subscribe(self, jid):
            print("[{}] Agent {} asked for subscription. Let's aprove it.".
→format(self.agent.name, jid.split("@")[0]))
            self.presence.approve(jid)
            self.presence.subscribe(jid)

        async def run(self):
            self.presence.set_available()
            self.presence.on_subscribe = self.on_subscribe
            self.presence.on_subscribed = self.on_subscribed
            self.presence.on_available = self.on_available


if __name__ == "__main__":

    jid1 = input("Agent1 JID> ")
    passwd1 = getpass.getpass()

    jid2 = input("Agent2 JID> ")
    passwd2 = getpass.getpass()

    agent2 = Agent2(jid2, passwd2)
    agent1 = Agent1(jid1, passwd1)
    agent1.jid2 = jid2
    agent2.jid1 = jid1
    agent2.start()
```

```python
agent1.start()

while True:
    try:
        time.sleep(1)
    except KeyboardInterrupt:
        break
agent1.stop()
agent2.stop()
```
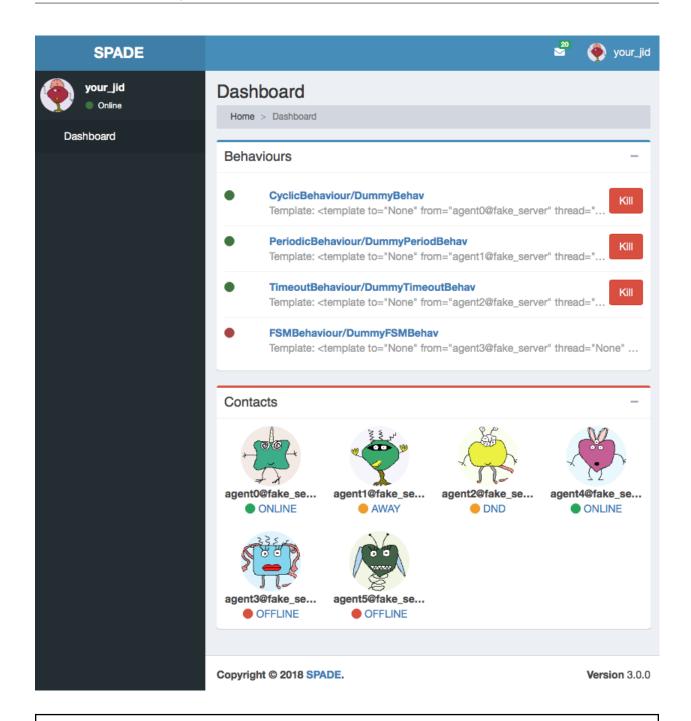
Web Graphical Interface

Each agent in SPADE provides a graphical interface *by default* that is accesible via web under the `/spade` path. To activate the web interface you just have to start the web module of the agent just as follows:

```python
agent = MyAgent("your_jid@your_xmpp_server", "your_password")
agent.start()
agent.web.start(hostname="127.0.0.1", port="10000")
```

Then you can open a web browser and go to the url `http://127.0.0.1:10000/spade` and you'll see the main page of your agent:
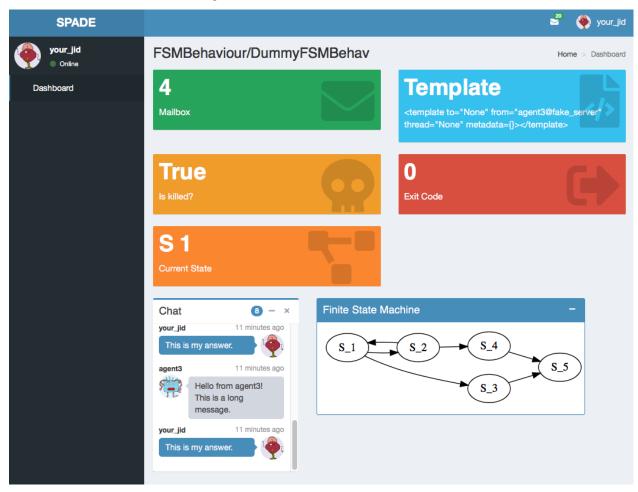
Warning: Remember to change the example's jids and passwords by your own accounts. These accounts do not exist and are only for demonstration purposes.

In the previous image you can see the index page of an agent, where you can check its name and avatar, a list of its behaviours, and a list of its contacts. In the top menu bar you can also check its incoming messages and the profile menu of the agent where you can stop the agent.

Caution: Note that if you run several agents with the web interface each agent **should** have a different port in order to avoid errors because some port is busy. The hostname can also be customized if you need to expose only
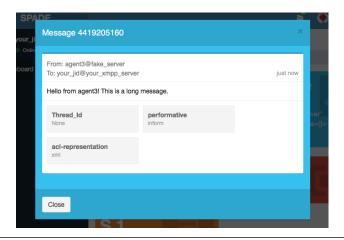
> to `localhost` or to a public ip (or even `0.0.0.0`).

In the behaviours box you can see all the behaviours that have been added to the agent, both the active ones and the ended ones. You can click the **kill** button to stop a behaviour and you can click the behaviour's name to see more information about it as in the next image:
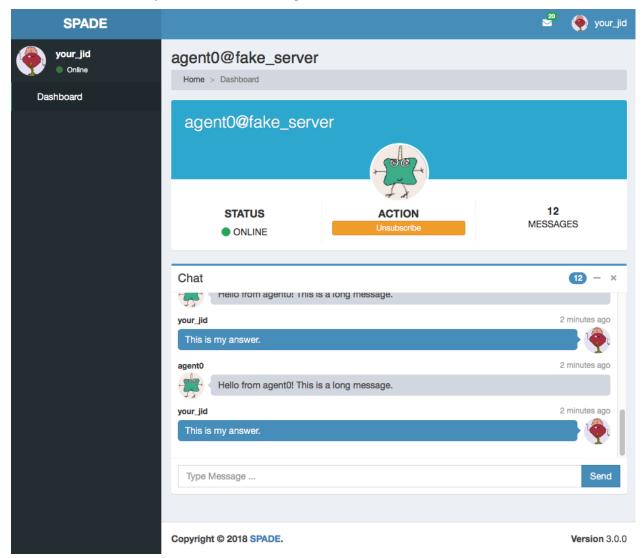

In this page you can check some important information about the behaviour such as its mailbox, the template with which it was registered, wether if it is killed or not or its exit code. Also, each type of behaviour may show any information related with its internal data, e.g. in the previous image you can see that the behaviour is an FSMBehaviour and the interface shows the current state where the FSM is and an image with the structure of the FSM. Finally, you can also check all the messages that have been sent or received from/to this behaviour in the chat box.

Note: To see more information about a message just click on the message text and you'll see something like the next image:

From the index page you can also click on any of your contacts to see information about them. In a contact's page you can check the presence status of your contact, unsubscribe from it and check the messages exchanged with it in the chat box. In such chat box you can also send a message to the contact.

9.1 Creating custom web interfaces

SPADE web module can also be used to create your own applications served by your agents themselves. You can register new paths in the web module and, following the model-view-controller paradigm (MVC), register controllers that compute the necessary data from the agent (the model) and render a template (the view) which will be served when someone requests the path with which it was registered. Let's see an example:

```python
async def hello_controller(request):
    return {"number": 42}

a = Agent("your_jid@your_xmpp_server", "your_password")

a.web.add_get("/hello", hello_controller, "hello.html")

a.start(auto_register=True)
a.web.start(port=10000)
```

In this example there are some elements that must be explained:

1. The `hello_controller` function is a coroutine (see the `async` statement) that returns a dictionary with data that will be rendered in the template.

2. The `add_get` method allows us to register a new controller with a path and a template.

3. You can alternatively use the `add_post` method if the http request must be of type **POST** (e.g. sending forms).

Next we are going to explain a little more about the controller, the path and the template.

Note: Please, do not use the `/spade` path o avoid conflicts with the default agent pages (unless you want to modify them).

9.1.1 Controller

The controller is the asyncronous method (or coroutine) that prepares the data to render the web page. It is an `async` method that always receives a single argument: `request`. A controller queries the model, which in our case is the agent (accesible in your coroutines using `self.agent`) and prepares a dictionary which will be used to render the template (as we will see in a moment). Inside a controller coroutine you can do any agent related stuff (sending messages, starting or killing behaviours, etc.).

Hint: Just remember the trick that a coroutine should not be too intensive in cpu, to avoid blocking the execution of the agent.

An example of controller would be:

```python
async def my_behaviours_controller(request):
    behaviours_list = []
    for b in self.agent.behaviours:
        behaviours_list.append(str(b))

    return {
            "behaviours": behaviours_list,
            "rand": random.random()
            }
```

This controller would pass a variable called "behaviours" with the names of all the agent's behaviours to the template, which would be able to render such data. It also generates a random number which is stored in the "rand" key of the data passed to the template.

In the case that your controller responds to a **POST** query (it has been registered with the add_post method) you can recover the data sent through the request by using this snippet of code:

```python
async def my_post_controller(request):
    form = await request.post()
```

Following the last example, the *form* variable would be a dictionary containing the data sent from the client. This way you can create forms in your web applications to be sent to your agents.

Hint: Instead of returning a dict with data to be rendered you can redirect to another URL by raising an HTTPFound exception from the aiohttp.web module as in the next example:

```python
from aiohttp import web

async def my_redirect_controller(request):
    raise web.HTTPFound("/")
```

JSON Responses

In the case that you need to return a JSON Response instead of an HTML page, is as simple as follows: call the add_get or add_post method passing None as the template argument. Thus, the dictionary that you are returning in your controller coroutine will be built into a JSON Response instead of rendering a jinja2 template.

Example:

```python
async def json_controller(self, request):
    return {"my_data": {'a': 0, 'b': 1, 'c': 2}}

self.web.add_get("/home", self.json_controller, template=None)
```

Hint: You may also use the raw=True parameter in the add_get and add_post methods to indicate that the returned result should not be processed neither by jinja2 nor json parsing.

9.1.2 Path

The path will define where your application will respond to requests. You can use any allowed character for defining paths. To define variable paths you can also use the aiohttp syntax. For example, a path /a/{agentjid}/c would match with the url /a/agent@server/c. Then, in your controller, you can recover the agentjid value using the request object:

```python
async def my_controller(request):
    jid = request.match_info['agentjid']
    return {"jid": jid}
```

9.1.3 Template

The template is an HTML file with an specific format which allows you to prepare dynamic web pages that are rendered with the information generated by your agent. The SPADE templates are created in the Jinja2 format, which allows the rendering process to have variables that come from our agent controllers and control structures.

In Jinja 2 variables are wrapped with double curly brackets (e.g. `{{ my_variable }}`) and the control structures with curly brackets and the percentage symbol (e.g. `{% if my_variable %} Hello World {% endif %}`).

Note: To know more about the Jinja 2 template engine please visit: http://jinja.pocoo.org/docs/

Hint: To allow SPADE to find your templates you can use the `templates_path` argument when starting the web module:

```
agent.web.start(port=10000, templates_path="static/templates")
```

A simple example of template would be:

```html
<html>
  <head>
    <title>{{ agent.jid }}</title>
  </head>
  <body>
      My favourite number is {{ number }}
      <h2>My behaviours:</h2>
      <ol>
      {% for b in behaviours %}
          <li> {{ b }} </li>
      {% endfor %}
      </ol>
  </body>
</html>
```

Note: Note that the `agent` variable is always available in your templates to help you to access your internal data.

Extending SPADE with plugins

This release of SPADE is designed as a very light version of the platform (compared with SPADE<3.0) which provides only the core features that a MAS platform should have. This implies that some of the features that were provided by previous versions of the platform are now not included.

How makes that sense? Well, all that previous features are not lost, but are going to be turned into plugins that you can connect to your MAS application.

This way it is very easy to add new features to SPADE without disturbing the core development.

We have planned three different ways to design plugins for the SPADE platform, but of course we are open to suggestions.

> **Warning:** A plugin needs to comply with some requirements to be accepted as a SPADE plugin and be listed as an official plugin on the main page:
>
> 1. It must be open source (of course!) and published in PyPi.
>
> 2. The package must be called spade-* (e.g.: spade-bdi, spade-owl, etc.) and be imported as `import spade_*`.
>
> 3. It must be tested.
>
> 4. It must follow the PEP8.

You can develop *new behaviours*, *new mixins* that modify behaviours, and of course *new libraries* that your agents can use inside your behaviours. Let's see some examples of each of these ones:

10.1 New Behaviours

Developing new behaviours is as easy as creating a new class that inherits from `spade.behaviour.CyclicBehaviour` (or any of its subclassed behaviours) and overload the methods that are needed. Pay atention to the methods that are related with the control flow of a behaviour like `_step`, `done` and `_run`. And remember that you *should not* overload the methods that are reserved for the user to be overloaded: `on_start`, `run` and `on_end`.

Example:

```python
class BDIBehaviour(spade.behaviour.PeriodicBehaviour):

    async def _step(self):
        # the bdi stuff

    def add_belief(self, ...):
        ...
    def add_desire(self, ...):
        ...
    def add_intention(self, ...):
        ...
    def done(self):
        # the done evaluation

    ...
```

10.2 New Mixins

Some cases you don't want to add a new behaviour, but to add new features to current behaviours or agents. This can be done by means of *mixins*. A mixin is a class that a behaviour or an agent can inherit from, in addition to the original parent class, making use of the multiple inheritance of python. This way, when we are creating our agent and we implement its behaviour which is (for example) a cyclic behaviour and we want to add this behaviour a feature that is provided by a plugin called spade-p2p that allows the agent to send P2P messages (by modifying the send and receive methods of the behaviour) we should do the following:

```python
from spade_p2p import P2PMixin

class MyNewBehaviour(P2PMixin, CyclicBehaviour):
    ...
    async def run(self):
        ...
        self.send(my_message, p2p=True)
        ...
```

> **Warning:** The order of your mixins is important! The base behaviour class **must** be **always** the last one in the method resolution order.

> **Hint:** Remember that if you need to call the parent function of the base behaviour (or any other mixin in the method resolution order), you must use the super() function (see the following example).

To develop this example mixin you should do the following:

```python
class P2PMixin(object):
    async def send(self, msg, p2p=False):
        if p2p:
            await self.send_p2p(msg)
        else:
            await super().send(msg)
```

```python
    async def send_p2p(self, msg):
        ...
```

In case you need to apply the mixin to the `Agent` class there are two hook coroutines that are prepared to be overriden if needed. These coroutines are `_hook_plugin_before_connection` and `_hook_plugin_after_connection`. They will be called before and after the connection to the server is done respectively. In order to support multiple mixins it is **important** to call always to the parent method. Next, an example of how to build a simple mixin is shown:

```python
class MyMixin:
    async def _hook_plugin_before_connection(self, *args, **kwargs):
        await super()._hook_plugin_before_connection(*args, **kwargs)
        # do my plugin stuff before the connection is done

    async def _hook_plugin_after_connection(self, *args, **kwargs):
        await super()._hook_plugin_after_connection(*args, **kwargs)
        # do my plugin stuff after the connection is done

class MyAgent(MyMixin, Agent):
    # Inherit always from mixins first. Last class to inherit from is Agent.
```

10.3 New Libraries

Finally, the easiest way to add new features to your agents is by means of *libraries*. If you want your agents to support, for example, the OWL content language, you don't need to change spade, just make a library that handles it. Example:

```python
from spade_owl import parse as owl_parse
from spade_owl import dump as owl_dump

class MyBehaviour(spade.behaviour.CyclicBehaviour):
    async def run(self):
        msg = await self.receive()

        owl_content = owl_parse(msg.content)
        # do wat you want with the owl content

        reply.content = owl_dump(...my owl reply...)

        await self.send(reply)
```

API Documentation

11.1 spade package

11.1.1 Submodules

11.1.2 spade.agent module

class spade.agent.**Agent**(*jid: str*, *password: str*, *verify_security: bool = False*)

    Bases: `object`

    add_behaviour(*behaviour: Type[spade.behaviour.CyclicBehaviour]*, *template: Optional[spade.template.Template] = None*) → None

        Adds and starts a behaviour to the agent. If template is not None it is used to match new messages and deliver them to the behaviour.

        Args: behaviour (Type[spade.behaviour.CyclicBehaviour]): the behaviour to be started template (spade.template.Template, optional): the template to match messages with (Default value = None)

    avatar

        Generates a unique avatar for the agent based on its JID. Uses Gravatar service with MonsterID option.

        Returns: str: the url of the agent's avatar

    static build_avatar_url(*jid: str*) → str

        Static method to build a gravatar url with the agent's JID

        Args: jid (aioxmpp.JID): an XMPP identifier

        Returns: str: an URL for the gravatar

    dispatch(*msg: spade.message.Message*) → List[_asyncio.Future]

        Dispatch the message to every behaviour that is waiting for it using their templates match.

        Args: msg (spade.message.Messagge): the message to dispatch.

        Returns: list(asyncio.Future): a list of futures of the append of the message at each matched behaviour.

get (*name: str*) → Any
> Recovers a knowledge item from the agent's knowledge base.

> **Args:** name(str): name of the item

> **Returns:** object: the object retrieved or None

has_behaviour (*behaviour: Type[spade.behaviour.CyclicBehaviour]*) → bool
> Checks if a behaviour is added to an agent.

> **Args:** behaviour (Type[spade.behaviour.CyclicBehaviour]): the behaviour instance to check

> **Returns:** bool: a boolean that indicates wether the behaviour is inside the agent.

is_alive () → bool
> Checks if the agent is alive.

> **Returns:** bool: wheter the agent is alive or not

name
> Returns the name of the agent (the string before the '@')

> **Returns:** str: the name of the agent (the string before the '@')

remove_behaviour (*behaviour: Type[spade.behaviour.CyclicBehaviour]*) → None
> Removes a behaviour from the agent. The behaviour is first killed.

> **Args:** behaviour (Type[spade.behaviour.CyclicBehaviour]): the behaviour instance to be removed

set (*name: str*, *value: Any*)
> Stores a knowledge item in the agent knowledge base.

> **Args:** name (str): name of the item value (object): value of the item

set_container (*container: spade.container.Container*) → None
> Sets the container to which the agent is attached

> **Args:** container (spade.container.Container): the container to be attached to

set_loop (*loop*) → None

setup () → None
> Setup agent before startup. This coroutine may be overloaded.

start (*auto_register: bool = True*) → Union[Coroutine, _asyncio.Future]
> Tells the container to start this agent. It returns a coroutine or a future depending on whether it is called from a coroutine or a synchronous method.

> **Args:** auto_register (bool): register the agent in the server (Default value = True)

> **Returns:** Coroutine: if called from an async method Future: if called from a synchronized method

stop () → Union[Coroutine, _asyncio.Future]
> Tells the container to start this agent. It returns a coroutine or a future depending on whether it is called from a coroutine or a synchronous method.

submit (*coro: Coroutine*) → _asyncio.Future
> Runs a coroutine in the event loop of the agent. this call is not blocking.

> **Args:** coro (Coroutine): the coroutine to be run

> **Returns:** asyncio.Future: the future of the coroutine execution

exception spade.agent.**AuthenticationFailure**
> Bases: Exception

11.1.3 spade.behaviour module

exception spade.behaviour.**BehaviourNotFinishedException**

Bases: Exception

class spade.behaviour.**CyclicBehaviour**

Bases: object

This behaviour is executed cyclically until it is stopped.

enqueue(*message: spade.message.Message*) → None

Enqueues a message in the behaviour's mailbox

Args: message (spade.message.Message): the message to be enqueued

exit_code

Returns the exit_code of the behaviour. It only works when the behaviour is done or killed, otherwise it raises an exception.

Returns: object: the exit code of the behaviour

Raises: BehaviourNotFinishedException: if the behaviour is not yet finished

get(*name: str*) → Any

Recovers a knowledge item from the agent's knowledge base.

Args: name (str): name of the item

Returns: Any: the object retrieved or None

is_done() → bool

Check if the behaviour is finished

Returns: bool: whether the behaviour is finished or not

is_killed() → bool

Checks if the behaviour was killed by means of the kill() method.

Returns: bool: whether the behaviour is killed or not

join(*timeout: Optional[float] = None*) → Optional[Coroutine]

Wait for the behaviour to complete

Args: timeout (Optional[float]): an optional timeout to wait to join (if None, the join is blocking)

Returns: None: if called from a synchronous method Coroutine: if called from an async method

Raises: TimeoutError: if the timeout is reached

kill(*exit_code: Optional[Any] = None*) → None

Stops the behaviour

Args: exit_code (object, optional): the exit code of the behaviour (Default value = None)

mailbox_size() → int

Checks if there is a message in the mailbox

Returns: int: the number of messages in the mailbox

match(*message: spade.message.Message*) → bool

Matches a message with the behaviour's template

Args: message(spade.message.Message): the message to match with

Returns: bool: wheter the messaged matches or not

on_end() → None
    Coroutine called after the behaviour is done or killed.

on_start() → None
    Coroutine called before the behaviour is started.

receive(*timeout: Optional[float] = None*) → Optional[spade.message.Message]
    Receives a message for this behaviour. If timeout is not None it returns the message or "None" after timeout is done.

    Args: timeout (float, optional): number of seconds until return

    Returns: spade.message.Message: a Message or None

run() → None
    Body of the behaviour. To be implemented by user.

send(*msg: spade.message.Message*) → None
    Sends a message.

    Args: msg (spade.message.Message): the message to be sent.

set(*name: str*, *value: Any*) → None
    Stores a knowledge item in the agent knowledge base.

    Args: name (str): name of the item value (Any): value of the item

set_agent(*agent*) → None
    Links behaviour with its owner agent

    Args: agent (spade.agent.Agent): the agent who owns the behaviour

set_template(*template: spade.template.Template*) → None
    Sets the template that is used to match incoming messages with this behaviour.

    Args: template (spade.template.Template): the template to match with

start() → None
    starts behaviour in the event loop

class spade.behaviour.**FSMBehaviour**
    Bases: *spade.behaviour.CyclicBehaviour*

    A behaviour composed of states (oneshotbehaviours) that may transition from one state to another.

    add_state(*name: str*, *state: spade.behaviour.State*, *initial: bool = False*) → None
        Adds a new state to the FSM.

        Args: name (str): the name of the state, which is used as its identifier. state (spade.behaviour.State): The state class initial (bool, optional): wether the state is the initial state or not. (Only one initial state is allowed) (Default value = False)

    add_transition(*source: str*, *dest: str*) → None
        Adds a transition from one state to another.

        Args: source (str): the name of the state from where the transition starts dest (str): the name of the state where the transition ends

    get_state(*name*) → spade.behaviour.State

    get_states() → Dict[str, spade.behaviour.State]

    is_valid_transition(*source: str*, *dest: str*) → bool
        Checks if a transitions is registered in the FSM

        Args: source (str): the source state name dest (str): the destination state name

Returns: bool: wether the transition is valid or not

run() → None
> In this kind of behaviour there is no need to overload run. The run methods to be overloaded are in the State class.

setup() → None

to_graphviz() → str
> Converts the FSM behaviour structure to Graphviz syntax

> **Returns:** str: the graph in Graphviz syntax

exception spade.behaviour.**NotValidState**
> Bases: Exception

exception spade.behaviour.**NotValidTransition**
> Bases: Exception

class spade.behaviour.**OneShotBehaviour**
> Bases: *spade.behaviour.CyclicBehaviour*

> This behaviour is only executed once

class spade.behaviour.**PeriodicBehaviour**(*period:* *float*, *start_at:* *Optional[datetime.datetime] = None*)
> Bases: *spade.behaviour.CyclicBehaviour*

> This behaviour is executed periodically with an interval

> **period**
> > Get the period.

class spade.behaviour.**State**
> Bases: *spade.behaviour.OneShotBehaviour*

> A state of a FSMBehaviour is a OneShotBehaviour

> **set_next_state**(*state_name:* *str*) → None
> > Set the state to transition to when this state is finished. state_name must be a valid state and the transition must be registered. If set_next_state is not called then current state is a final state.

> > **Args:** state_name (str): the name of the state to transition to

class spade.behaviour.**TimeoutBehaviour**(*start_at*)
> Bases: *spade.behaviour.OneShotBehaviour*

> This behaviour is executed once at after specified datetime

spade.behaviour.**now**()
> Returns new datetime object representing current time local to tz.

> > **tz** Timezone object.

> If no tz is specified, uses local timezone.

11.1.4 spade.container module

class spade.container.**AioThread**(**args*, ***kwargs*)
> Bases: threading.Thread

> **finalize**() → None

---

run() → None
> Method representing the thread's activity.

> You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class spade.container.**Container**
> Bases: *spade.container.Container*

> The container class allows agents to exchange messages without using the XMPP socket if they are in the same process. The container is a singleton.

spade.container.**stop_container**() → None

11.1.5 spade.message module

class spade.message.**Message**(*to: Optional[str] = None*, *sender: Optional[str] = None*, *body: Optional[str] = None*, *thread: Optional[str] = None*, *metadata: Optional[Dict[str, str]] = None*)
> Bases: *spade.message.MessageBase*

> **make_reply**() → spade.message.Message
> > Creates a copy of the message, exchanging sender and receiver

> > **Returns:** spade.message.Message: a new message with exchanged sender and receiver

> **prepare**() → aioxmpp.stanza.Message
> > Returns an aioxmpp.stanza.Message built from the Message and prepared to be sent.

> > **Returns:** aioxmpp.stanza.Message: the message prepared to be sent

class spade.message.**MessageBase**(*to: Optional[str] = None*, *sender: Optional[str] = None*, *body: Optional[str] = None*, *thread: Optional[str] = None*, *metadata: Optional[Dict[str, str]] = None*)
> Bases: object

> **body**
> > Get body of the message Returns:

> > > str: the body of the message

> **classmethod from_node**(*node: aioxmpp.stanza.Message*) → Type[spade.message.MessageBase]
> > Creates a new spade.message.Message from an aixoxmpp.stanza.Message

> > **Args:** node (aioxmpp.stanza.Message): an aioxmpp Message

> > **Returns:** spade.message.Message: a new spade Message

> **get_metadata**(*key: str*) → str
> > Get the value of a metadata. Returns None if metadata does not exist.

> > **Args:** key (str): name of the metadata

> > **Returns:** str: the value of the metadata (or None)

> **id**

> **match**(*message: Type[MessageBase]*) → bool
> > Returns wether a message matches with this message or not. The message can be a Message object or a Template object.

> > **Args:** message (spade.message.Message): the message to match to

Returns: bool: wether the message matches or not

sender
Get jid of the sender

Returns: aioxmpp.JID: jid of the sender

set_metadata(*key: str*, *value: str*) → None
Add a new metadata to the message

Args: key (str): name of the metadata value (str): value of the metadata

thread
Get Thread of the message

Returns: str: thread id

to
Gets the jid of the receiver.

Returns: aioxmpp.JID: jid of the receiver

11.1.6 spade.presence module

exception spade.presence.**ContactNotFound**
Bases: Exception

class spade.presence.**PresenceManager**(*agent*)
Bases: object

approve(*peer_jid: str*) → None
Approve a subscription request from jid

Args: peer_jid (str): the JID to approve

get_contact(*jid: aioxmpp.structs.JID*) → Dict
Returns a contact

Args: jid (aioxmpp.JID): jid of the contact

Returns: dict: the roster of contacts

get_contacts() → Dict[str, Dict]
Returns list of contacts

Returns: dict: the roster of contacts

is_available() → bool
Returns the available flag from the state

Returns: bool: wether the agent is available or not

on_available(*peer_jid: str*, *stanza: aioxmpp.stanza.Presence*) → None
Callback called when a contact becomes available. To ve overloaded by user.

Args: peer_jid (str): the JID of the agent that is available stanza (aioxmpp.Presence): The presence message containing type, show, priority and status values.

on_subscribe(*peer_jid: str*) → None
Callback called when a subscribe query is received. To ve overloaded by user.

Args: peer_jid (str): the JID of the agent asking for subscription

on_subscribed(*peer_jid: str*) → None
Callback called when a subscribed message is received. To ve overloaded by user.

Args: peer_jid (str): the JID of the agent that accepted subscription

on_unavailable(*peer_jid: str*, *stanza: aioxmpp.stanza.Presence*) → None
Callback called when a contact becomes unavailable. To ve overloaded by user.

Args: peer_jid (str): the JID of the agent that is unavailable stanza (aioxmpp.Presence): The presence message containing type, show, priority and status values.

on_unsubscribe(*peer_jid: str*) → None
Callback called when an unsubscribe query is received. To ve overloaded by user.

Args: peer_jid (str): the JID of the agent asking for unsubscription

on_unsubscribed(*peer_jid: str*) → None
Callback called when an unsubscribed message is received. To ve overloaded by user.

Args: peer_jid (str): the JID of the agent that unsubscribed

priority
The currently set priority which is broadcast when the client connects and when the presence is re-emitted.

This attribute cannot be written. It does not reflect the actual presence seen by others. For example when the client is in fact offline, others will see unavailable presence no matter what is set here.

Returns: int: the priority of the connection

set_available(*show: Optional[<unknown>.PresenceShow] = <PresenceShow.NONE: None>*)
Sets the agent availability to True.

Args: show (aioxmpp.PresenceShow, optional): the show state of the presence (Default value = PresenceShow.NONE)

set_presence(*state: Optional[aioxmpp.structs.PresenceState] = None*, *status: Optional[str] = None*,
priority: Optional[int] = None)
Change the presence broadcast by the client. If the client is currently connected, the new presence is broadcast immediately.

Args: state(aioxmpp.PresenceState, optional): New presence state to broadcast (Default value = None) status(dict or str, optional): New status information to broadcast (Default value = None) priority (int, optional): New priority for the resource (Default value = None)

set_unavailable() → None
Sets the agent availability to False.

state
The currently set presence state (as aioxmpp.PresenceState) which is broadcast when the client connects and when the presence is re-emitted.

This attribute cannot be written. It does not reflect the actual presence seen by others. For example when the client is in fact offline, others will see unavailable presence no matter what is set here.

Returns: aioxmpp.PresenceState: the presence state of the agent

status
The currently set textual presence status which is broadcast when the client connects and when the presence is re-emitted.

This attribute cannot be written. It does not reflect the actual presence seen by others. For example when the client is in fact offline, others will see unavailable presence no matter what is set here.

Returns: dict: a dict with the status in different languages (default key is None)

subscribe(*peer_jid: str*) → None
Asks for subscription

Args: peer_jid (str): the JID you ask for subscriptiion

unsubscribe(*peer_jid: str*) → None
    Asks for unsubscription

    Args: peer_jid (str): the JID you ask for unsubscriptiion

11.1.7 spade.template module

class spade.template.**ANDTemplate**(*expr1*, *expr2*)
    Bases: *spade.template.BaseTemplate*

    match(*message*)

class spade.template.**BaseTemplate**
    Bases: object

    Template operators

class spade.template.**NOTTemplate**(*expr*)
    Bases: *spade.template.BaseTemplate*

    match(*message*)

class spade.template.**ORTemplate**(*expr1*, *expr2*)
    Bases: *spade.template.BaseTemplate*

    match(*message*)

class spade.template.**Template**(*to: Optional[str] = None*, *sender: Optional[str] = None*, *body: Optional[str] = None*, *thread: Optional[str] = None*, *metadata: Optional[Dict[str, str]] = None*)
    Bases: *spade.template.BaseTemplate*, *spade.message.MessageBase*

    Template for message matching

class spade.template.**XORTemplate**(*expr1*, *expr2*)
    Bases: *spade.template.BaseTemplate*

    match(*message*)

11.1.8 spade.trace module

class spade.trace.**TraceStore**(*size: int*)
    Bases: object

    Stores and allows queries about events.

    all(*limit: Optional[int] = None*) → List[spade.message.Message]
        Returns all the events, until a limit if defined

        Args: limit (int, optional): the max length of the events to return (Default value = None)

        Returns: list: a list of events

    append(*event: spade.message.Message*, *category: Optional[str] = None*) → None
        Adds a new event to the trace store. The event may hava a category

        Args: event (spade.message.Message): the event to be stored category (str, optional): a category to classify the event (Default value = None)

filter(*limit: Optional[int] = None*, *to: Optional[str] = None*, *category: Optional[str] = None*) →
List[spade.message.Message]
Returns the events that match the filters

Args: limit (int, optional): the max length of the events to return (Default value = None) to (str, optional):
only events that have been sent or received by 'to' (Default value = None) category (str, optional):
only events belonging to the category (Default value = None)

Returns: list: a list of filtered events

len() → int
Length of the store

Returns: int: the size of the trace store

received(*limit: Optional[int] = None*) → List[spade.message.Message]
Returns all the events that have been received (excluding sent events), until a limit if defined

Args: limit (int, optional): the max length of the events to return (Default value = None)

Returns: list: a list of received events

reset() → None
Resets the trace store

11.1.9 spade.web module

class spade.web.**WebApp**(*agent*)
Bases: object

Module to handle agent's web interface

add_get(*path: str*, *controller: Coroutine*, *template: str*, *raw: Optional[bool] = False*) → None
Setup a route of type GET

Args: path (str): URL to listen to controller (coroutine): the coroutine to handle the request template
(str): the template to render the response or None if it is a JSON response raw (bool): indicates if
post-processing (jinja, json, etc) is needed or not

add_post(*path: str*, *controller: Coroutine*, *template: str*, *raw: Optional[bool] = False*) → None
Setup a route of type POST

Args: path (str): URL to listen to controller (coroutine): the coroutine to handle the request template
(str): the template to render the response or None if it is a JSON response raw (bool): indicates if
post-processing (jinja, json, etc) is needed or not

agent_processor(*request*)

find_behaviour(*behaviour_str: str*) → Optional[Type[spade.behaviour.CyclicBehaviour]]

get_agent(*request*)

get_behaviour(*request*)

get_messages(*request*)

index(*request*)

is_started() → bool

kill_behaviour(*request*)

send_agent(*request*)

setup_routes() → None

start (*hostname: Optional[str] = None*, *port: Optional[int] = None*, *templates_path: Optional[str] = None*) → None
Starts the web interface.

Args: hostname (str, optional): host name to listen from. (Default value = None) port (int, optional): port to listen from. (Default value = None) templates_path (str, optional): path to look for templates. (Default value = None)

stop_agent (*request*)

stop_now (*request*)

static timeago (*date*)

unsubscribe_agent (*request*)

spade.web.**start_server_in_loop** (*runner: aiohttp.web_runner.AppRunner*, *hostname: str*, *port: int*, *agent*)
Listens to http requests and sends them to the webapp.

Args: runner (AppRunner): AppRunner to process the http requests hostname (str): host name to listen from. port (int): port to listen from. agent (spade.agent.Agent): agent that owns the web app.

spade.web.**unused_port** (*hostname: str*) → None
Return a port that is unused on the current host.

11.1.10 Module contents

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

12.1 Types of Contributions

12.1.1 Implement Plugins

SPADE can be extended by means of plugins. See how to develop one at *Extending SPADE with plugins*.

12.1.2 Report Bugs

Report bugs at https://github.com/javipalanca/spade/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

12.1.3 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

12.1.4 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

12.1.5 Write Documentation

SPADE could always use more documentation, whether as part of the official SPADE docs, in docstrings, or even on the web in blog posts, articles, and such.

12.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/javipalanca/spade/issues.

If you are proposing a feature:

- Explain in detail how it would work.

- Keep the scope as narrow as possible, to make it easier to implement.

- Remember that this is a volunteer-driven project, and that contributions are welcome :)

12.2 Get Started!

Ready to contribute? Here's how to set up *spade* for local development.

1. Fork the *spade* repo on GitHub.

2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/spade.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv spade
$ cd spade/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 spade tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

12.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.

2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

3. The pull request should work for Python 3.6, and for PyPy. Check https://travis-ci.org/javipalanca/spade/pull_requests and make sure that the tests pass for all supported Python versions.

12.4 Tips

To run a subset of tests:

```
$ py.test tests.test_agent
```

Code of Conduct

13.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

13.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

13.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

13.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

13.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at jpalanca AT dsic DOT upv DOT es. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

13.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant homepage, version 1.4.

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

Credits

14.1 Development Lead

- Javi Palanca <https://github.com/javipalanca>

14.2 Contributors

- Sergio Alemany <https://github.com/Gersiete>

History

15.1 3.2.2 (2021-11-25)

- Hotfix for the event loop in windows when python is 3.6

15.2 3.2.1 (2021-11-16)

- Fixed event loop for windows

15.3 3.2.0 (2021-07-13)

- Added support for Python 3.8 and 3.9
- Fixed support for Linux, Windows and macOS

15.4 3.1.9 (2021-07-08)

- Minor fix in docs.

15.5 3.1.8 (2021-07-08)

- Added examples.
- Fixed documentation examples.
- Added Github CI support.

15.6 3.1.7 (2021-06-25)

- Added hooks for plugins.
- Minor bug fixings.

15.7 3.1.6 (2020-05-22)

- Fixed coverage and ci.

15.8 3.1.5 (2020-05-21)

- Fixed how to stop behaviours.
- Fixed some tests.
- Blackstyled code.

15.9 3.1.4 (2019-11-04)

- Fixed issue with third party versions.
- Use factories in tests.
- Updated documentation and examples.
- Minor bug fixing.

15.10 3.1.3 (2019-07-18)

- Added BDI plugin (https://github.com/javipalanca/spade_bdi).
- Improved the platform stop (quit_spade).
- Minor bug fixing.

15.11 3.1.2 (2019-05-14)

- Hotfix docs.

15.12 3.1.1 (2019-05-14)

- Added Python 3.7 support.
- Added Code of Conduct.
- Minor bugs fixed.

15.13 3.1.0 (2019-03-22)

- Agents now run in a single event loop managed by the container.
- Behaviors can be waited for using the "join" method.
- To check if a behaviours is done you can now use the "is_done" method.
- The "setup" method is now a coroutine.
- New "quit_spade" helper to stop the whole process.
- The "start" and "stop" methods change depending on the context, since it is the container who will properly start or stop the agent. They return a coroutine or a future depending on whether they are called from a coroutine or a synchronous method.

15.14 3.0.9 (2018-10-24)

- Added raw parameter to allow raw web responses.
- Changed default agent urls to the "/spade" namespace to avoid conflicts.

15.15 3.0.8 (2018-10-02)

- Added a container mechanism to speedup local sends.
- Added performance example.
- Improved API doc.
- Added container tests.

15.16 3.0.7 (2018-09-27)

- Fixed bug when running FSM states.
- Improved Message __str__.
- Fixed bug when thread is not defined in a message.
- aioxmpp send method is now in client instead of stream.

15.17 3.0.6 (2018-09-27)

- Added statement to relinquish the cpu at each behaviour loop.
- Message Thread is now stored as metadata for simplicity.

15.18 3.0.5 (2018-09-21)

- Added JSON responses in web module.
- Some improvements in aiothread management.

15.19 3.0.4 (2018-09-20)

- Added coroutines to start agents from within other agents.
- Improved API doc format.

15.20 3.0.3 (2018-09-12)

- Rename internal templates to avoid conflicts.
- Added API doc.
- Minor bug fixes.

15.21 3.0.2 (2018-09-12)

- Fixed presence notification updates.
- Fixed FSM graphviz visualization.
- Raise AuthenticationFailure Exception when user is not registered or user or password is wrong.
- Import init improvements.
- Attribute auto_register is now default True.
- Improved documentation.
- Other minor fixes.

15.22 3.0.1 (2018-09-07)

- Minor doc fixings and improvements.

15.23 3.0.0 (2017-10-06)

- Started writing 3.0 version.

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

A

add_behaviour() (*spade.agent.Agent method*), [49](#)
add_get() (*spade.web.WebApp method*), [58](#)
add_post() (*spade.web.WebApp method*), [58](#)
add_state() (*spade.behaviour.FSMBehaviour method*), [52](#)
add_transition() (*spade.behaviour.FSMBehaviour method*), [52](#)
Agent (*class in spade.agent*), [49](#)
agent_processor() (*spade.web.WebApp method*), [58](#)
AioThread (*class in spade.container*), [53](#)
all() (*spade.trace.TraceStore method*), [57](#)
ANDTemplate (*class in spade.template*), [57](#)
append() (*spade.trace.TraceStore method*), [57](#)
approve() (*spade.presence.PresenceManager method*), [55](#)
AuthenticationFailure, [50](#)
avatar (*spade.agent.Agent attribute*), [49](#)

B

BaseTemplate (*class in spade.template*), [57](#)
BehaviourNotFinishedException, [51](#)
body (*spade.message.MessageBase attribute*), [54](#)
build_avatar_url() (*spade.agent.Agent static method*), [49](#)

C

ContactNotFound, [55](#)
Container (*class in spade.container*), [54](#)
CyclicBehaviour (*class in spade.behaviour*), [51](#)

D

dispatch() (*spade.agent.Agent method*), [49](#)

E

enqueue() (*spade.behaviour.CyclicBehaviour method*), [51](#)

exit_code (*spade.behaviour.CyclicBehaviour attribute*), [51](#)

F

filter() (*spade.trace.TraceStore method*), [57](#)
finalize() (*spade.container.AioThread method*), [53](#)
find_behaviour() (*spade.web.WebApp method*), [58](#)
from_node() (*spade.message.MessageBase class method*), [54](#)
FSMBehaviour (*class in spade.behaviour*), [52](#)

G

get() (*spade.agent.Agent method*), [49](#)
get() (*spade.behaviour.CyclicBehaviour method*), [51](#)
get_agent() (*spade.web.WebApp method*), [58](#)
get_behaviour() (*spade.web.WebApp method*), [58](#)
get_contact() (*spade.presence.PresenceManager method*), [55](#)
get_contacts() (*spade.presence.PresenceManager method*), [55](#)
get_messages() (*spade.web.WebApp method*), [58](#)
get_metadata() (*spade.message.MessageBase method*), [54](#)
get_state() (*spade.behaviour.FSMBehaviour method*), [52](#)
get_states() (*spade.behaviour.FSMBehaviour method*), [52](#)

H

has_behaviour() (*spade.agent.Agent method*), [50](#)

I

id (*spade.message.MessageBase attribute*), [54](#)
index() (*spade.web.WebApp method*), [58](#)
is_alive() (*spade.agent.Agent method*), [50](#)
is_available() (*spade.presence.PresenceManager method*), [55](#)
is_done() (*spade.behaviour.CyclicBehaviour method*), [51](#)

77