Agent Handlers

Introduction

Within the uAgents Framework, you can allow agents to perform specific actions at the moment some sort of condition has been satisfied. You can do so, you will need to use a specific handler. These are like decorators and are identified by the following code syntax:.on_... .

Below, we show how to use the following different event handlers:

1. Interval tasks
2. :.on_interval()
3. Handle messages
4. :.on_message()
5. Answer queries
6. :.on_query()

Creating an interval task with on_interval() handler

Sometimes an agent will need to perform a task periodically. To do this we can use theon_interval() decorator which periodically repeats a given function for the agent. For instance, an agent could send a message every 2 seconds to another agent.

Let's get started and create our first interval task!

Walk-through

1. Let's create a Python script for this task, and name it by running:touch interval-task.py
2. Then import the necessary classes fromuagents
3. library,Agent
4. andContext
5. , and create our agent:
6. from
7. uagents
8. import
9. Agent
10. ,
11. Context
12. agent
13. =
14. Agent
15. (name
16. =
17. "alice"
18. , seed
19. =
20. "alice recovery phrase"
21.)
22. Create a function to handle the startup event, which will introduce the agent:
23. @agent
24. .
25. on_event
26. (
27. "startup"
28.)
29. async
30. def
31. introduce_agent
32. (
33. ctx
34. :
35. Context):
36. ctx
37. .
38. logger
39. .
40. info

41. (
42. f
43. "Hello, I'm agent
44. {
45. agent.name
46. }
47. and my address is
48. {
49. agent.address
50. }
51. ."
52.)
53. We can now define our agent's interval behavior. We want our agent to log a message every 2 seconds using theon_interval
54. decorator:
55. @agent
56. .
57. on_interval
58. (period
59. =
60. 2.0
61.)
62. async
63. def
64. say_hello
65. (
66. ctx
67. :
68. Context):
69. ctx
70. .
71. logger
72. .
73. info
74. (
75. "Hello!"
76.)
77. if
78. **name**
79. ==
80. "**main**"
81. :
82. agent
83. .
84. run
85. ()
86. The output will be printed out using thectx.logger.info()
87. method.
88. Save the script.

The overall script should look as follows:

interval-task.py from uagents import Agent , Context

agent

Agent (name = "alice" , seed = "alice recovery phrase")

@agent . on_event ("startup") async

def

introduce_agent (ctx : Context): ctx . logger . info (f "Hello, I'm agent { agent.name } and my address is { agent.address } .")

@agent . on_interval (period = 2.0) async

def

say_hello (ctx : Context): ctx . logger . info ("Hello!")

if

name

==

"**main**" : agent . run ()

Run the script

Run the script:python interval-task.py

The output should be as follows:

hello, my name is alice hello, my name is alice hello, my name is alice

Handle messages using the on_message() handler

We now showcase a scenario where three agents, namedalice ,bob , andcharles , use a customprotocol ↗ to communicate. In the example, Alice and Bob support the protocol, whereas Charles attempts to send broadcast messages to all agents using the protocol. Agents use theon_message() handler which allows them to handle messages matching specific data models.

Let's get started!

Walk-through

1. First of all, let's create a Python script for this task, and name it:touch broadcast.py
2. We then need to import theAgent
3. ,Bureau
4. ,Context
5. ,Model
6. , andProtocol
7. classes from theuagents
8. library, and thefund_agent_if_low
9. fromuagents.setup
10. . Then, let's create the 3 different agents using the classAgent
11. . Each agent is initialized with a unique name and a seed phrase for wallet recovery. Additionally, if an agent's wallet balance is low, thefund_agent_if_low()
12. function is called to add funds to their wallet:
13. from
14. uagents
15. import
16. Agent
17. ,
18. Bureau
19. ,
20. Context
21. ,
22. Model
23. ,
24. Protocol
25. from
26. uagents
27. .
28. setup
29. import
30. fund_agent_if_low
31. alice
32. =
33. Agent
34. (name
35. =
36. "alice"
37. , seed
38. =
39. "alice recovery phrase"

)
bob
=
Agent
(name
=
"bob"
, seed
=
"bob recovery phrase"
)
charles
=
Agent
(name
=
"charles"
, seed
=
"charles recovery phrase"
)
fund_agent_if_low
(alice.wallet.
address
())
fund_agent_if_low
(bob.wallet.
address
())
fund_agent_if_low
(charles.wallet.
address
())
It is optional but useful to include aseed
parameter when creating an agent to set fixedaddresses ↗
. Otherwise, random addresses will be generated every time you run the agent.
Let's then define the message data models to specify the type of messages being handled and exchanged by the agents. We define aBroadcastExampleRequest
and aBroadcastExampleResponse
data models. Finally, create aprotocol
namedproto
with version1.0
:
class
BroadcastExampleRequest
(
Model
):
pass
class
BroadcastExampleResponse
(
Model
):
text
:
str
proto
=
Protocol
(name
=
"proto"
, version
=
"1.0"
)
Let's now define a message handler function for incoming messages of typeBroadcastExampleRequest

in the protocol:
@proto
.
on_message
(model
=
BroadcastExampleRequest, replies
=
BroadcastExampleResponse)
async
def
handle_request
(
ctx
:
Context
,
sender
:
str
,
_msg
:
BroadcastExampleRequest):
await
ctx
.
send
(
sender,
BroadcastExampleResponse
(text
=
f
"Hello from
{
ctx.agent.name
}
"
)
)
Here we defined ahandle_request()
function which is used whenever a request is received. This sends a response back to the sender. This function is decorated with the.on_message()
decorator indicating that this function is triggered whenever a message of typeBroadcastExampleRequest
is received. The function sends a response containing a greeting message with the name of the agent that sent the request in the first place.
Now, we need to include theprotocol
into the agents. Specifically, the protocol is included in bothalice
andbob
agents. This means they will follow the rules defined in the protocol when communicating:
alice
.
include
(proto)
bob
.
include
(proto)
ℹ
After the first registration in theAlmanac ↗
smart contract, it will take about 5 minutes before the agents can be found through the protocol.
It is now time to define the behavior and function ofcharles
agent:
@charles
.
on_interval
(period

```
=
5
)
async
def
say_hello
(
ctx
:
Context):
status_list
=
await
ctx
.
broadcast
(proto.digest, message
=
BroadcastExampleRequest
())
ctx
.
logger
.
info
(
f
"Trying to contact
{
len
(status_list)
}
agents."
)
@charles
.
on_message
(model
=
BroadcastExampleResponse)
async
def
handle_response
(
ctx
:
Context
,
sender
:
str
,
msg
:
BroadcastExampleResponse):
ctx
.
logger
.
info
(
f
"Received response from
{
sender
}
:
{
```

241. msg.text
242. }
243. "
244.)
245. In the first part, we use the.on_interval()
246. decorator to define an interval behavior for this agent when the script is being run. In this case, the agent will execute thesay_hello()
247. function every 5 seconds. TheContext
248. object is a collection of data and functions related to the agent. Inside thesay_hello()
249. function, the agent uses thectx.broadcast()
250. method to send a broadcast message. The message is of typeBroadcastExampleRequest()
251. and it is being sent using the protocol's digest (proto.digest
252.).
253. Then, we defined a.on_message()
254. decorator which decorateshandle_response()
255. function. This function handles all incoming messages of typeBroadcastExampleResponse
256. from other agents. When a response is received, it logs the information. Inside thehandle_response()
257. function, the agent logs an informational message usingctx.logger.info()
258. method to print the sender and the content of the message. The message includes the sender's name and the text content of the response message.
259. We are now ready to set up aBureau
260. object for agents to be run together at the same time, and we addalice
261. ,bob
262. , andcharles
263. to it using thebureau.add()
264. method:
265. bureau
266. =
267. Bureau
268. (port
269. =
270. 8000
271. , endpoint
272. =
273. "http://localhost:8000/submit"
274.)
275. bureau
276. .
277. add
278. (alice)
279. bureau
280. .
281. add
282. (bob)
283. bureau
284. .
285. add
286. (charles)
287. if
288. **name**
289. ==
290. "**main**"
291. :
292. bureau
293. .
294. run
295. ()
296. The bureau is assigned to listen onport=8000
297. and specifies anendpoint
298. at"http://localhost:8000/submit"
299. for submitting data.
300. Save the script.

The overall script should look as follows:

broadcast.py from uagents import Agent , Bureau , Context , Model , Protocol

create agents

alice and bob will support the protocol

charles will try to reach all agents supporting the protocol

alice

Agent (name = "alice" , seed = "alice recovery phrase") bob =

Agent (name = "bob" , seed = "bob recovery phrase") charles =

Agent (name = "charles" , seed = "charles recovery phrase")

class

BroadcastExampleRequest (Model): pass

class

BroadcastExampleResponse (Model): text :

str

define protocol

proto

Protocol (name = "proto" , version = "1.0")

@proto . on_message (model = BroadcastExampleRequest, replies = BroadcastExampleResponse) async

def

handle_request (ctx : Context ,

sender :

str ,

_msg : BroadcastExampleRequest): await ctx . send (sender, BroadcastExampleResponse (text = f "Hello from { ctx.agent.name } "))

include protocol

Note: after the first registration on the almanac smart contract, it will

take about 5 minutes before the agents can be found through the protocol

alice . include (proto) bob . include (proto)

let charles send the message to all agents supporting the protocol

@charles . on_interval (period = 5) async

def

say_hello (ctx : Context): status_list =

await ctx . broadcast (proto.digest, message = BroadcastExampleRequest ()) ctx . logger . info (f "Trying to contact { len (status_list) } agents.")

@charles . on_message (model = BroadcastExampleResponse) async

def

handle_response (ctx : Context ,

sender :

str ,

msg : BroadcastExampleResponse): ctx . logger . info (f "Received response from { sender } : { msg.text } ")

bureau

Bureau (port = 8000 , endpoint = "http://localhost:8000/submit") bureau . add (alice) bureau . add (bob) bureau . add (charles)

if

name

==

"**main**" : bureau . run ()

Run the script

Make sure to have activated your virtual environment correctly.

Run the script:python broadcast.py

The output would be:

Trying to contact 2 agents. Received response from alice: Hello from alice Received response from bob: Hello from bob

Answer queries with on_query() handler

Theon_query() handler is used to register aFunction ↗ as a handler for incoming queries that match a specifiedModel . This decorator enables the agent to respond to queries in an event-driven manner.

Walk-through

Agent's script

For the agent, the script sets up an agent to handle incoming queries. It defines two models:TestRequest andResponse . Upon startup, it logs the agent's details. The core functionality lies in thequery_handler , decorated with@agent.on_query() , which processes received queries and sends back a predefined response. This demonstrates creating responsive agents within theuagents Framework, showcasing how they can interact with other agents or functions in an asynchronous, event-driven architecture.

agent.py from uagents import Agent , Context , Model

class

TestRequest (Model): message :

str

class

Response (Model): text :

str

Initialize the agent with its configuration.

agent

Agent (name = "your_agent_name_here" , seed = "your_agent_seed_here" , port = 8001 , endpoint = "http://localhost:8001/submit" ,)

@agent . on_event ("startup") async

def

startup (ctx : Context): ctx . logger . info (f "Starting up { agent.name } ") ctx . logger . info (f "With address: { agent.address } ") ctx . logger . info (f "And wallet address: { agent.wallet. address () } ")

Decorator to handle incoming queries.

@agent . on_query (model = TestRequest, replies = {Response}) async

def

query_handler (ctx : Context ,

sender :

str ,

_query : TestRequest): ctx . logger . info ("Query received") try :

do something here

await ctx . send (sender, Response (text = "success")) except

Exception : await ctx . send (sender, Response (text = "fail"))

Main execution block to run the agent.

if

name

==

"**main**" : agent . run () The agent is created using theAgent class fromuagents library. It is initialised with aname ,seed ,port , andendpoint . It defines anon_event() handler for thestartup event, where it logs information about the agent's initialisation. It defines anon_query() handler for handling queries of typeTestRequest . Upon receiving a query, it processes it and sends back aResponse . The agent is then set to run.

Proxy

The proxy is implemented usingFastAPI . It sets up two routes:"/" for a simple root message and"/endpoint" for receiving requests. When aPOST request is made to"/endpoint" with a JSON payload containing aTestRequest , it triggers themake_agent_call function. Insidemake_agent_call , it callsagent_query to communicate with the agent. The agent receives the query, processes it, and sends back a response. The proxy receives the response from the agent and sends back a success message along with the response text.

Let's explore the Proxy code script step-by-step:

1. First of all navigate to directory where you want to create your project.
2. Create a Python script nameon_query.py
3. by runningtouch on_query.py
4. .
5. We need to importjson
6. ,asyncio
7. ,uagent
8. 'sModel

andquery
. Then we would need to define the query format using theQueryRequest
class as a subclass ofModel
:
import
json
from
fastapi
import
FastAPI
from
uagents
import
Model
from
uagents
.
query
import
query
AGENT_ADDRESS
=
"agent1qt6ehs6kqdgtrsduuzslqnrzwkrcn3z0cfvwsdj22s27kvatrxu8sy3vag0"
class
TestRequest
(
Model
):
message
:
str
Createagent_query()
function to send query to agent and decode the response received.
async
def
agent_query
(
req
):
response
=
await
query
(destination
=
AGENT_ADDRESS, message
=
req, timeout
=
15.0
)
data
=
json
.
loads
(response.
decode_payload
())
return
data
[
"text"
]
Initialize a FastAPI app:
app
=
FastAPI

77. ()
78. Define a root endpoint to test the server:
79. @app
80. .
81. get
82. (
83. "/"
84.)
85. def
86. read_root
87. ():
88. return
89. "Hello from the Agent controller"
90. Define an endpoint to make agent calls:
91. @app
92. .
93. post
94. (
95. "/endpoint"
96.)
97. async
98. def
99. make_agent_call
100. (
101. req
102. :
103. TestRequest):
104. try
105. :
106. res
107. =
108. await
109. agent_query
110. (req)
111. return
112. f
113. "successful call - agent response:
114. {
115. res
116. }
117. "
118. except
119. Exception
120. :
121. return
122. "unsuccessful agent call"
123. Save the script.

The overall script should look as follows:

on_query.py import json

from fastapi import FastAPI from uagents import Model from uagents . query import query

AGENT_ADDRESS

"agent1qt6ehs6kqdgtrsduuzslqnrzwkrcn3z0cfvwsdj22s27kvatrxu8sy3vag0"

class

TestRequest (Model): message :

str

async

def

agent_query (req): response =

await

query (destination = AGENT_ADDRESS, message = req, timeout = 15.0) data = json . loads (response. decode_payload ()) return data ["text"]

app

FastAPI ()

@app . get ("/") def

read_root (): return

"Hello from the Agent controller"

@app . post ("/endpoint") async

def

make_agent_call (req : TestRequest): try : res =

await

agent_query (req) return

f "successful call - agent response: { res } " except

Exception : return

"unsuccessful agent call"

Run the example

In separate terminals:

1. Run theFastAPI proxy
2. :uvicorn proxy:app
3. Run theagent
4. :python agent.py
5. Query the agent via the proxy:curl -d '{"message": "test"}' -H "Content-Type: application/json" -X POST http://localhost:8000/endpoint

Was this page helpful?

Communicating with other agents Agents storage functions