How to use the agents to simulate a cleaning scenario

Introduction

In this guide we will show how to set up protocols and different AI Agents to implement a cleaning service between clients and cleaning service providers using theuagents library.

What you need

For this example, you need to have installed multiple libraries. Below, you can find the ones you need and related commands to install them directly from the terminal:

1. uagents ↗
2. .
3. tortoise
4. :pip install tortoise-orm
5. .
6. geopy
7. :pip install geopy
8. .
9. pytz
10. :pip install pytz
11. .

Walk-through

1. First of all, you need to navigate towards the directory you created for your project and create a folder for this task:mkdir cleaning_demo
2. .
3. Inside this folder we will create another folder for our protocols:mkdir protocols
4. .
5. Within theprotocols
6. folder, let's create a sub-folder:mkdir cleaning
7. . This is the folder including all of the scripts for our cleaning protocol to run correctly.
8. After having defined our protocols folder, we need to create 2 scripts, one for ouruser
9. and the other one forcleaner
10. agents.

We can start by writing the code for our protocol.

Cleaning Protocol

TheCleaning Protocol is made up of two scripts**:init**.py andmodels.py .

Models

First, we need to define amodels protocol which defines the structure and relationships of users, service types, providers, and availability within the cleaning service application. The defined models and their relationships will be used for data storage, retrieval, and manipulation throughout the application's logic.

1. First of all, navigate towards the cleaning directory and create a Python script for this task and name it by running:touch models.py
2. .
3. We then import the necessary classes fromenum
4. andtortoise
5. libraries, and then define the different data models:
6. models.py
7. from
8. enum
9. import
10. Enum
11. from
12. tortoise
13. import

fields
,
models
class
ServiceType
(
int
,
Enum
):
FLOOR
=
1
WINDOW
=
2
LAUNDRY
=
3
IRON
=
4
BATHROOM
=
5
class
User
(
models
.
Model
):
id
=
fields
.
IntField
(pk
=
True
)
name
=
fields
.
CharField
(max_length
=
64
)
address
=
fields
.
CharField
(max_length
=
100
)
created_at
=
fields
.
DatetimeField
(auto_now_add
=
True
)

```
class
Service
(
models
.
Model
):
id
=
fields
.
IntField
(pk
=
True
)
type
=
fields
.
IntEnumField
(ServiceType)
class
Provider
(
models
.
Model
):
id
=
fields
.
IntField
(pk
=
True
)
name
=
fields
.
CharField
(max_length
=
64
)
location
=
fields
.
CharField
(max_length
=
64
)
created_at
=
fields
.
DatetimeField
(auto_now_add
=
True
)
availability
=
fields
```

```
.
ReverseRelation
[
"Availability"
]
services
=
fields
.
ManyToManyField
(
"models.Service"
)
markup
=
fields
.
FloatField
(default
=
1.1
)
class
Availability
(
models
.
Model
):
id
=
fields
.
IntField
(pk
=
True
)
provider
=
fields
.
OneToOneField
(
"models.Provider"
, related_name
=
"availability"
)
max_distance
=
fields
.
IntField
(default
=
10
)
time_start
=
fields
.
DatetimeField
()
time_end
=
fields
.
```

DatetimeField
()
min_hourly_price
=
fields
.
FloatField
(default
=
0.0
)
We defined the following data models:

- ServiceType
- is an enumeration class that defines different types of cleaning services. Each service type is associated with an integer value. Service types includeFLOOR
- ,WINDOW
- ,LAUNDRY
- ,IRON
- , andBATHROOM
- .
- User
- is a model representing the users of the cleaning service. It takes in the following fields:
  - id
  - : Auto-incrementing integer field serving as the primary key.
  - name
  - : Character field with a maximum length of 64 for the user's name.
  - address
  - : Character field with a maximum length of 100 for the user's address.
  - created_at
  - : DateTime field automatically set to the current timestamp when the user is created.
- Service
- is model representing types of cleaning services. It takes in the following fields:
  - id
  - : Auto-incrementing integer field serving as the primary key.
  - type

- : Enum field storing the ServiceType enumeration values.
- Provider
- is a model representing cleaning service providers. It considers the following fields:
- id
- : Auto-incrementing integer field serving as the primary key.
- name
- : Character field with a maximum length of 64 for the provider's name.
- location
- : Character field with a maximum length of 64 for the provider's location.
- created_at
- : DateTime field automatically set to the current timestamp when the provider is created.
- availability
- : Reverse relation to theAvailability
- model.
- services
- : Many-to-many relationship with theService
- model.
- markup
- : Float field representing the markup applied to service prices.
- Availability
- is a model representing the availability of a cleaning service provider. It considers the following fields:
- id
- : Auto-incrementing integer field serving as the primary key.
- provider

276.
    - : One-to-one relationship to theProvider
277.
    - model.
278.
    - max_distance
279.
    - : Integer field representing the maximum distance a provider can travel for services.
280.
    - time_start
281.
    - : DateTime field indicating the start time of availability.
282.
    - time_end
283.
    - : DateTime field indicating the end time of availability.
284.
    - min_hourly_price
285.
    - : Float field representing the minimum hourly price for services.
286. Save the script.

Now that we have defined the protocols for this application, we proceed by defining the user and cleaner agents.

init

Now let's define a protocol handling communication between a user requesting a cleaning service and a cleaning service provider.

1. Inside the cleaning directory, create a Python script for this task and name it by running:touch**init**.py
2. .
3. Install geopy usingpip install geopy
4. in your terminal.(Geopy helps to know details about location and also calculate distance between two places.)
5. Let's import the necessary classes fromdatetime
6. ,typing
7. ,geopy.geocoders
8. ,geopy.distance
9. ,uagents
10. , and.models
11. . Let's then define the message models and handlers for the service request, service response, service booking, and booking response using theuagents
12. library. We then proceed and create a cleaning_proto object using the agentsProtocol
13. class. We give it the namecleaning
14. and the version0.1.0
15. . This protocol will be used to define handlers and manage communication between agents using the defined message models:
16. from
17. datetime
18. import
19. datetime
20. ,
21. timedelta
22. from
23. typing
24. import
25. List
26. from
27. geopy
28. .

```
geocoders
import
Nominatim
from
geopy
.
distance
import
geodesic
from
uagents
import
Context
,
Model
,
Protocol
from
.
models
import
Provider
,
Availability
,
User
PROTOCOL_NAME
=
"cleaning"
PROTOCOL_VERSION
=
"0.1.0"
class
ServiceRequest
(
Model
):
user
:
str
location
:
str
time_start
:
datetime
duration
:
timedelta
services
:
List
[
int
]
max_price
:
float
class
ServiceResponse
(
Model
):
accept
:
bool
price
:
```

float
class
ServiceBooking
(
Model
):
location
:
str
time_start
:
datetime
duration
:
timedelta
services
:
List
[
int
]
price
:
float
class
BookingResponse
(
Model
):
success
:
bool
cleaning_proto
=
Protocol
(name
=
PROTOCOL_NAME, version
=
PROTOCOL_VERSION)
We would then need to define anin_service_region
function, which determines whether a user's location is within the service area of a cleaning service provider. This function uses thegeopy
library for geolocation-related calculations and comparisons:
def
in_service_region
(
location
:
str
,
availability
:
Availability
,
provider
:
Provider
)
->
bool
:
geolocator
=
Nominatim
(user_agent
=
"micro_agents"

```
)
user_location
=
geolocator
.
geocode
(location)
cleaner_location
=
geolocator
.
geocode
(provider.location)
if
user_location
is
None
:
raise
RuntimeError
(
f
"user location
{
location
}
not found"
)
if
cleaner_location
is
None
:
raise
RuntimeError
(
f
"provider location
{
provider.location
}
not found"
)
cleaner_coordinates
=
(cleaner_location
.
latitude
,
cleaner_location
.
longitude)
user_coordinates
=
(user_location
.
latitude
,
user_location
.
longitude)
service_distance
=
geodesic
(user_coordinates, cleaner_coordinates).
miles
in_range
=
```

service_distance
<=
availability
.
max_distance
return
in_range
This function takes in the following parameters:

- location
- : The location of the user as a string (e.g., an address).
- availability
- : An instance of theAvailability
- class that holds information about the provider's availability.
- provider
- : An instance of theProvider
- class that holds information about the cleaning service provider.

The function starts by creating a geolocator object from theNominatim
class with a user agent stringmicro_agents
. Thegeolocator
is then used to obtain the geographical coordinates (latitude and longitude) of both the user's location and the provider's location. The function checks whether the geocoding for both locations was successful. If either location is not found, it raises aRuntimeError
with a relevant error message. The geographical coordinates of the user's location and the provider's location are used to calculate the distance (in miles) between them using thegeodesic
function from thegeopy.distance
module. The calculated distance is stored in theservice_distance
variable. Finally, the function compares the calculatedservice_distance
with the maximum distance allowed for the provider's availability: if the calculated distance is less than or equal to the provider's maximum distance, the user's location is considered within the service range, and thein_range
variable is set toTrue
. Otherwise, thein_range
variable is set toFalse
. The function returns the value ofin_range
, which indicates whether the user's location is within the service area of the provider.
We then need to define an event handler using the@cleaning_proto.on_message
decorator. This handler is responsible for processing incoming service requests, evaluates various conditions, and generates appropriate responses indicating whether the request is accepted and proposing a price if applicable. It takes inServiceRequest
messages and generatesServiceResponse
messages as responses:
@cleaning_proto
.
on_message
(model
=
ServiceRequest, replies
=
ServiceResponse)
async
def
handle_query_request
(
ctx
:
Context
,
sender
:
str
,

msg
:
ServiceRequest):
provider
=
await
Provider
.
filter
(name
=
ctx.name).
first
()
availability
=
await
Availability
.
get
(provider
=
provider)
services
=
[
int
(service.type)
for
service
in
await
provider
.
services]
markup
=
provider
.
markup
user
,
_
=
await
User
.
get_or_create
(name
=
msg.user, address
=
sender)
msg_duration_hours
:
float
=
msg
.
duration
.
total_seconds
()
/
3600
ctx
.
logger

```
.
info
(
f
"Received service request from user `
{
user.name
}
`"
)
if
(
set
(msg.services)
<=
set
(services)
and
in_service_region
(msg.location, availability, provider)
and
availability
.
time_start
<=
msg
.
time_start
and
availability
.
time_end

        =

msg
.
time_start
+
msg
.
duration
and
availability
.
min_hourly_price
*
msg_duration_hours
<
msg
.
max_price
)
:
accept
=
True
price
=
markup
*
availability
.
min_hourly_price
*
msg_duration_hours
ctx
.
logger
```

.
info
(
f
"I am available! Proposing price:
{
price
}
."
)
else
:
accept
=
False
price
=
0
ctx
.
logger
.
info
(
"I am not available. Declining request."
)
await
ctx
.
send
(sender,
ServiceResponse
(accept
=
accept, price
=
price))
We defined a@cleaning_proto.on_message()
decorator. Within the decorator, we defined thehandle_query_request()
function which is the actual event handler that gets executed when aServiceRequest
message is received. It takes the following arguments:ctx
,sender
, andmsg
.
The handler first retrieves the provider's information, availability, services, and markup using the context's information. It uses theUser.get_or_create
method to get or create a user instance based on the user's name and sender address. The function then calculates the message duration in hours. The handler then checks multiple conditions to decide whether the service request can be accepted:

- Whether the requested services are within the provider's available services.
- Whether the user's location is within the provider's service area using thein_service_region
- function.
- Whether the requested time is within the provider's availability.
- Whether the calculated price based on the minimum hourly price and markup is within the user's maximum price.

If all conditions are met, the handler setsaccept
toTrue
and calculates the proposedprice
based on the markup and availability's minimum hourly price. If any condition is not met, the handler setsaccept
toFalse
and theprice
to0
. Finally, the handler sends aServiceResponse
message back to the sender with the calculatedaccept

flag andprice
. The handler logs relevant information using thectx.logger.info()
method.
We then define another event handler,handle_book_request()
function, using the@cleaning_proto.on_message()
decorator. This handler processes incoming booking requests, evaluates various conditions, and generates appropriate responses indicating whether the service request was successful. It is triggered whenever aServiceBooking
message is received and generates aBookingResponse
messages as response:
@cleaning_proto
.
on_message
(model
=
ServiceBooking, replies
=
BookingResponse)
async
def
handle_book_request
(
ctx
:
Context
,
sender
:
str
,
msg
:
ServiceBooking):
provider
=
await
Provider
.
filter
(name
=
ctx.name).
first
()
availability
=
await
Availability
.
get
(provider
=
provider)
services
=
[
int
(service.type)
for
service
in
await
provider
.
services]
user
=
await

```
User
.
get
(address
=
sender)
msg_duration_hours
:
float
=
msg
.
duration
.
total_seconds
()
/
3600
ctx
.
logger
.
info
(
f
"Received booking request from user `
{
user.name
}
`"
)
success
=
(
set
(msg.services)
<=
set
(services)
and
availability
.
time_start
<=
msg
.
time_start
and
availability
.
time_end

        =

msg
.
time_start
+
msg
.
duration
and
msg
.
price
<=
availability
.
min_hourly_price
```

*
msg_duration_hours
)
if
success
:
availability
.
time_start
=
msg
.
time_start
+
msg
.
duration
await
availability
.
save
()
ctx
.
logger
.
info
(
"Accepted task and updated availability."
)

send the response

await
ctx
.
send
(sender,
BookingResponse
(success
=
success))
We defined the@cleaning_proto.on_message()
decorator which is triggered when aServiceBooking
message is received. The event handlerhandle_book_request()
function gets executed when aServiceBooking
message is received. It takes the following parameters:ctx
,sender
, andmsg
. The handler starts by retrieving the provider's information, availability, and services using the context's information. It also retrieves the user instance based on the sender's address, and calculates the message duration in hours. The function then checks multiple conditions to decide whether the booking request can be accepted:

- Whether the requested services are within the provider's available services.
- Whether the requested time is within the provider's availability.
- Whether the requested price is within the acceptable range based on the availability's minimum hourly price and duration.

If the booking request meets the conditions for success, the handler updates the availability's start time to account for the duration of the service, and then saves the updated availability information. It then uses thectx.logger.info()
method to log that the task has been accepted and availability has been updated. Regardless of success, the handler sends aBookingResponse
message back to the sender with a flag indicating whether the booking request was successful or not.
Save the script.

The overall script should look as follows:

init.py from datetime import datetime , timedelta from typing import List

from geopy . geocoders import Nominatim from geopy . distance import geodesic

from uagents import Context , Model , Protocol from

. models import Provider , Availability , User

PROTOCOL_NAME

"cleaning" PROTOCOL_VERSION =

"0.1.0"

class

ServiceRequest (Model): user :

str location :

str time_start : datetime duration : timedelta services : List [int] max_price :

float

class

ServiceResponse (Model): accept :

bool price :

float

class

ServiceBooking (Model): location :

str time_start : datetime duration : timedelta services : List [int] price :

float

class

BookingResponse (Model): success :

bool

cleaning_proto

Protocol (name = PROTOCOL_NAME, version = PROTOCOL_VERSION)

def

in_service_region (location :

str ,

availability : Availability ,

provider : Provider) ->

bool : geolocator =

Nominatim (user_agent = "micro_agents")

user_location

geolocator . geocode (location) cleaner_location = geolocator . geocode (provider.location)

if user_location is

None : raise

RuntimeError (f "user location { location } not found")

if cleaner_location is

None : raise

RuntimeError (f "provider location { provider.location } not found")

cleaner_coordinates

(cleaner_location . latitude , cleaner_location . longitude) user_coordinates = (user_location . latitude , user_location . longitude)

service_distance

geodesic (user_coordinates, cleaner_coordinates). miles in_range = service_distance <= availability . max_distance

return in_range

@cleaning_proto . on_message (model = ServiceRequest, replies = ServiceResponse) async

def

handle_query_request (ctx : Context ,

sender :

str ,

msg : ServiceRequest): provider =

await Provider . filter (name = ctx.name). first () availability =

await Availability . get (provider = provider) services = [int (service.type)

for service in

await provider . services] markup = provider . markup

user , _ =

await User . get_or_create (name = msg.user, address = sender) msg_duration_hours :

float

= msg . duration . total_seconds ()

/

3600 ctx . logger . info (f "Received service request from user { user.name }")

if (set (msg.services)

<=

set (services) and

in_service_region (msg.location, availability, provider) and availability . time_start <= msg . time_start and availability . time_end

= msg . time_start + msg . duration and availability . min_hourly_price * msg_duration_hours < msg . max_price)
: accept =

True price = markup * availability . min_hourly_price * msg_duration_hours ctx . logger . info (f "I am available! Proposing price: { price } .") else : accept =

False price =

0 ctx . logger . info ("I am not available. Declining request.")

await ctx . send (sender, ServiceResponse (accept = accept, price = price))

@cleaning_proto . on_message (model = ServiceBooking, replies = BookingResponse) async

def

handle_book_request (ctx : Context ,

sender :

str ,

msg : ServiceBooking): provider =

await Provider . filter (name = ctx.name). first () availability =

await Availability . get (provider = provider) services = [int (service.type)

for service in

await provider . services]

user

await User . get (address = sender) msg_duration_hours :

float

= msg . duration . total_seconds ()

/

3600 ctx . logger . info (f "Received booking request from user{ user.name }")

success

(set (msg.services)

<=

set (services) and availability . time_start <= msg . time_start and availability . time_end

= msg . time_start + msg . duration and msg . price <= availability . min_hourly_price * msg_duration_hours)

if success : availability . time_start = msg . time_start + msg . duration await availability . save () ctx . logger . info ("Accepted task and updated availability.")

send the response

await ctx . send (sender, BookingResponse (success = success))

Cleaner agent

We are now ready to define ourcleaner agent.

1. Let's now create a Python script incleaning_demo
2. folder, and name it:touch cleaner.py
3. We now need to import the necessary classes and the protocols we previously defined, and then create ourcleaner
4. agent as an instance of theAgent
5. class and make sure it has enough funds to register within the Almanac contract by running thefund_agent_if_low()
6. function. The agent is configured with a specificname
7. ,port
8. ,seed
9. , andendpoint
10. :
11. from
12. datetime
13. import

```
datetime
from
pytz
import
utc
from
tortoise
import
Tortoise
from
protocols
.
cleaning
import
cleaning_proto
from
protocols
.
cleaning
.
models
import
Availability
,
Provider
,
Service
,
ServiceType
from
uagents
import
Agent
,
Context
from
uagents
.
setup
import
fund_agent_if_low
cleaner
=
Agent
(
name
=
"cleaner"
,
port
=
8001
,
seed
=
"cleaner secret phrase"
,
endpoint
=
{
"http://127.0.0.1:8001/submit"
: {},
},
)
fund_agent_if_low
(cleaner.wallet.
address
())
```

build the cleaning service agent from the cleaning protocol

cleaner
.
include
(cleaning_proto)
Thecleaner
agent includes the previously defined cleaning protocol (cleaning_proto
) using theinclude()
method. This integrates the protocol's models and handlers into the agent's capabilities. The agent is set up to interact with the cleaning service protocol, allowing it to communicate with users and providers according to the logic defined in the protocol's handlers.
We then define the behaviors of ourcleaner
agent:
@cleaner
.
on_event
(
"startup"
)
async
def
startup
(
_ctx
:
Context):
await
Tortoise
.
init
(
db_url
=
"sqlite://db.sqlite3"
, modules
=
{
"models"
: [
"protocols.cleaning.models"
]}
)
await
Tortoise
.
generate_schemas
()
provider
=
await
Provider
.
create
(name
=
cleaner.name, location
=
"London Kings Cross"
)
floor
=
await
Service

```
.
create
(type
=
ServiceType.FLOOR)
window
=
await
Service
.
create
(type
=
ServiceType.WINDOW)
laundry
=
await
Service
.
create
(type
=
ServiceType.LAUNDRY)
await
provider
.
services
.
add
(floor)
await
provider
.
services
.
add
(window)
await
provider
.
services
.
add
(laundry)
await
Availability
.
create
(
provider
=
provider,
time_start
=
utc.
localize
(datetime.
fromisoformat
(
"2022-01-31 00:00:00"
)),
time_end
=
utc.
localize
(datetime.
fromisoformat
(
```

"2023-05-01 00:00:00"
)),
max_distance
=
10
,
min_hourly_price
=
5
,
)
@cleaner
.
on_event
(
"shutdown"
)
async
def
shutdown
(
_ctx
:
Context):
await
Tortoise
.
close_connections
()
if
name
==
"**main**"
:
cleaner
.
run
()
Thestartup()
event handler function is decorated with the.on_event()
decorator. This handler is executed when thecleaner
agent starts up. Its purpose is to initialize the agent's environment, set up the database, and populate it with necessary data. This function initializes the Tortoise ORM (Object–Relational Mapping) with the specified database URL and modules. It prepares the ORM to work with the defined data models. The ORM is used to manage database connections and schema generation for the agent's data models. In the snippet above,db_url
is set tosqlite://db.sqlite3
, indicating the use of an SQLite database nameddb.sqlite3
.modules
specifies the module containing the data models related to the cleaning protocol. We then define theTortoise.generate_schemas()
which is a function that generates the database schemas based on the defined data models. It sets up the necessary tables and relationships in the database. Afterwards, we create instances of theProvider
andService
models and populates them with data.
AProvider
instance is created with thecleaner
agent's name ("cleaner") and location ("London Kings Cross").
ThreeService
instances are created with different service types:FLOOR
,WINDOW
, andLAUNDRY
. The createdService
instances are associated with theProvider
instance using theprovider.services.add()
method. This establishes a relationship between providers and the services they offer. We proceed by defining anAvailability
instance to represent the availability of the cleaning service provider: it includes details such as the provider, start and end times of availability, maximum service distance (10 miles), and minimum hourly price (5).

272. Save the script.

The overall code should look as follows:

cleaner.py from datetime import datetime from pytz import utc

from tortoise import Tortoise

from protocols . cleaning import cleaning_proto from protocols . cleaning . models import Availability , Provider , Service , ServiceType

from uagents import Agent , Context from uagents . setup import fund_agent_if_low

cleaner

Agent (name = "cleaner" , port = 8001 , seed = "cleaner secret phrase" , endpoint = { "http://127.0.0.1:8001/submit" : {}, },)

fund_agent_if_low (cleaner.wallet. address ())

build the cleaning service agent from the cleaning protocol

cleaner . include (cleaning_proto)

@cleaner . on_event ("startup") async

def

startup (_ctx : Context): await Tortoise . init (db_url = "sqlite://db.sqlite3" , modules = { "models" : ["protocols.cleaning.models"]}) await Tortoise . generate_schemas ()

provider

await Provider . create (name = cleaner.name, location = "London Kings Cross")

floor

await Service . create (type = ServiceType.FLOOR) window =

await Service . create (type = ServiceType.WINDOW) laundry =

await Service . create (type = ServiceType.LAUNDRY)

await provider . services . add (floor) await provider . services . add (window) await provider . services . add (laundry)

await Availability . create (provider = provider, time_start = utc. localize (datetime. fromisoformat ("2022-01-31 00:00:00")), time_end = utc. localize (datetime. fromisoformat ("2023-05-01 00:00:00")), max_distance = 10 , min_hourly_price = 5 ,)

@cleaner . on_event ("shutdown") async

def

shutdown (_ctx : Context): await Tortoise . close_connections ()

if

name

==

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

User

We are now ready to define ouruser agent.

1. Let's now create a Python script incleaning_demo
2. folder, and name it:touch user.py
3. We need to import the necessary classes and the protocols we previously defined, and then create ouruser
4. agent as an instance of theAgent
5. class and make sure it has enough funds to register within the Almanac contract by running thefund_agent_if_low()
6. function. The agent is configured with a specificname
7. ,port
8. ,seed
9. , andendpoint
10. . We also define thecleaner
11. 's address:
12. from
13. datetime
14. import
15. datetime
16. ,
17. timedelta
18. from
19. pytz
20. import
21. utc
22. from
23. protocols
24. .
25. cleaning
26. import
27. (
28. ServiceBooking
29. ,
30. BookingResponse
31. ,
32. ServiceRequest
33. ,
34. ServiceResponse
35. ,
36.)
37. from
38. protocols
39. .
40. cleaning
41. .
42. models
43. import
44. ServiceType
45. from
46. uagents
47. import
48. Agent
49. ,
50. Context
51. from
52. uagents
53. .
54. setup
55. import
56. fund_agent_if_low
57. CLEANER_ADDRESS
58. =
59. "agent1qdfdx6952trs028fxyug7elgcktam9f896ays6u9art4uaf75hwy2j9m87w"
60. user
61. =
62. Agent
63. (
64. name
65. =
66. "user"
67. ,
68. port

=
8000
,
seed
=
"cleaning user recovery phrase"
,
endpoint
=
{
"http://127.0.0.1:8000/submit"
: {},
},
)
fund_agent_if_low
(user.wallet.
address
())
Let's define aServiceRequest
object to represent a request for cleaning services. It includes information such as the user's name, location, start time, duration, types of services requested, and a maximum price. We also define aMARKDOWN
to apply a discount or markdown to the service:
request
=
ServiceRequest
(
user
=
user.name,
location
=
"London Kings Cross"
,
time_start
=
utc.
localize
(datetime.
fromisoformat
(
"2023-04-10 16:00:00"
)),
duration
=
timedelta
(hours
=
4
),
services
=
[ServiceType.WINDOW, ServiceType.LAUNDRY],
max_price
=
60
,
)
MARKDOWN
=
0.8
We then define an event handler to handle service requests:
@user
.
on_interval
(period
=
3.0
, messages

=
ServiceRequest)
async
def
interval
(
ctx
:
Context):
ctx
.
storage
.
set
(
"markdown"
, MARKDOWN)
completed
=
ctx
.
storage
.
get
(
"completed"
)
if
not
completed
:
ctx
.
logger
.
info
(
f
"Requesting cleaning service:
{
request
}
"
)
await
ctx
.
send
(CLEANER_ADDRESS, request)
Theinterval()
function is decorated with the.on_interval()
decorator indicating that is is executed at regular intervals specified by the period parameter to sendServiceRequest
messages. Within the function, thectx.storage.set()
method is called to store theMARKDOWN
constant in the context's storage under the key "markdown". The function first checks the value ofcompleted
which is retrieved from the context's storage usingctx.storage.get()
method. This is used to determine whether the service request has already been completed. If the service request is not marked as completed, an informational log message is generated usingctx.logger.info()
method indicating that a cleaning service request is being made. The{request}
placeholder in the log message is filled with the details of the request object previously created. Then,ctx.send()
method is called to send the cleaning service request to the specified address (CLEANER_ADDRESS
) alongside with the request object.
We then would need to define a function for handling responses to service queries and generating service bookings based on the received response:
@user
.
on_message
(ServiceResponse, replies

```
=
ServiceBooking)
async
def
handle_query_response
(
ctx
:
Context
,
sender
:
str
,
msg
:
ServiceResponse):
markdown
=
ctx
.
storage
.
get
(
"markdown"
)
if
msg
.
accept
:
ctx
.
logger
.
info
(
"Cleaner is available, attempting to book now"
)
booking
=
ServiceBooking
(
location
=
request.location,
time_start
=
request.time_start,
duration
=
request.duration,
services
=
request.services,
price
=
markdown
*
msg.price,
)
await
ctx
.
send
(sender, booking)
else
```

:
ctx
.
logger
.
info
(
"Cleaner is not available - nothing more to do"
)
ctx
.
storage
.
set
(
"completed"
,
True
)
Thehandle_query_response()
function is decorated with.on_message()
and handles messages of typeServiceResponse
. It also specifies that the function replies with instances ofServiceBooking
data model.
The function first retrieves the markdown value from the context's storage usingctx.storage.get()
method. If the receivedServiceResponse
message indicates acceptance, a log message is generated, indicating that a cleaner is available and an attempt to book the service will be made. AServiceBooking
object is created, containing information about the location, start time, duration, services, and calculated price (based on the received price in the response message and the markdown value). TheServiceBooking
object is then sent to the sender's address usingctx.send()
method.
If however the response message does not indicate acceptance, a log message is generated, indicating that a cleaner is not available. The "completed" key in the storage is set toTrue
, indicating that the service request process has been completed.
We finally need to define a function for handlingBookingResponse
messages:
@user
.
on_message
(BookingResponse, replies
=
set
())
async
def
handle_book_response
(
ctx
:
Context
,
_sender
:
str
,
msg
:
BookingResponse):
if
msg
.
success
:
ctx
.
logger
.

335. info
336. (
337. "Booking was successful"
338.)
339. else
340. :
341. ctx
342. .
343. logger
344. .
345. info
346. (
347. "Booking was UNSUCCESSFUL"
348.)
349. ctx
350. .
351. storage
352. .
353. set
354. (
355. "completed"
356. ,
357. True
358.)
359. if
360. **name**
361. ==
362. "**main**"
363. :
364. user
365. .
366. run
367. ()
368. Thehandle_book_response()
369. function is decorated with.on_message()
370. decorator indicating that it's meant to handle messages of type BookingResponse. However, it specifiesreplies=set()
371. , which means that this function is not intended to send any replies. The function processes the booking response message: if thesuccess
372. field in the message isTrue
373. , a log message is generated indicating that the booking was successful usingctx.logger.info()
374. method. If the success field isFalse
375. , a log message is generated indicating that the booking was unsuccessful. The "completed" key in the context's storage is set toTrue
376. using thectx.storage.set()
377. method to indicate that the entire booking process has been completed.
378. Save the script.

The overall script should look as follows:

user.py from datetime import datetime , timedelta from pytz import utc

from protocols . cleaning import (ServiceBooking , BookingResponse , ServiceRequest , ServiceResponse ,) from protocols . cleaning . models import ServiceType from uagents import Agent , Context from uagents . setup import fund_agent_if_low

CLEANER_ADDRESS

"agent1qdfdx6952trs028fxyug7elgcktam9f896ays6u9art4uaf75hwy2j9m87w"

user

Agent (name = "user" , port = 8000 , seed = "cleaning user recovery phrase" , endpoint = { "http://127.0.0.1:8000/submit" : {}, },)

fund_agent_if_low (user.wallet. address ())

request

ServiceRequest (user = user.name, location = "London Kings Cross" , time_start = utc. localize (datetime. fromisoformat ("2023-04-10 16:00:00")), duration = timedelta (hours = 4), services = [ServiceType.WINDOW, ServiceType.LAUNDRY], max_price = 60 ,)

MARKDOWN

0.8

@user . on_interval (period = 3.0 , messages = ServiceRequest) async

def

interval (ctx : Context): ctx . storage . set ("markdown" , MARKDOWN) completed = ctx . storage . get ("completed")

if

not completed : ctx . logger . info (f "Requesting cleaning service: { request } ") await ctx . send (CLEANER_ADDRESS, request)

@user . on_message (ServiceResponse, replies = ServiceBooking) async

def

handle_query_response (ctx : Context ,

sender :

str ,

msg : ServiceResponse): markdown = ctx . storage . get ("markdown") if msg . accept : ctx . logger . info ("Cleaner is available, attempting to book now") booking =

ServiceBooking (location = request.location, time_start = request.time_start, duration = request.duration, services = request.services, price = markdown * msg.price,) await ctx . send (sender, booking) else : ctx . logger . info ("Cleaner is not available - nothing more to do") ctx . storage . set ("completed" , True)

@user . on_message (BookingResponse, replies = set ()) async

def

handle_book_response (ctx : Context ,

_sender :

str ,

msg : BookingResponse): if msg . success : ctx . logger . info ("Booking was successful") else : ctx . logger . info ("Booking was UNSUCCESSFUL")

ctx . storage . set ("completed" , True)

if

name

==

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

Run the scripts

Run thecleaner and then theuser agents from different terminals:

- Terminal 1:python cleaner.py
- Terminal 2:python user.py

The output should be as follows, depending on the terminal:

- Cleaner
- [cleaner]: Received service request from user user
- [cleaner]: I am available! Proposing price: 22.0.
- [cleaner]: Received booking request from user user
- [cleaner]: Accepted task and updated availability.
- User
- [user]: Requesting cleaning service: user='user' location='London Kings Cross' time_start=datetime.datetime(2023, 4, 10, 16, 0, tzinfo=) duration=datetime.timedelta(seconds=14400) services=[,] max_price=60.0
- [user]: Cleaner is available, attempting to book now
- [user]: Booking was successful

Was this page helpful?

How to use agents to send tokens AI Agents: broadcast