

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:inits.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

```
14. fields
15. ,
16. models
17. class
18. ServiceType
19. (
20. int
21. ,
22. Enum
23. ):
24. FLOOR
25. =
26. 1
27. WINDOW
28. =
29. 2
30. LAUNDRY
31. =
32. 3
33. IRON
34. =
35. 4
36. BATHROOM
37. =
38. 5
39. class
40. User
41. (
42. models
43. .
44. Model
45. ):
46. id
47. =
48. fields
49. .
50. IntField
51. (pk
52. =
53. True
54. )
55. name
56. =
57. fields
58. .
59. CharField
60. (max_length
61. =
62. 64
63. )
64. address
65. =
66. fields
67. .
68. CharField
69. (max_length
70. =
71. 100
72. )
73. created_at
74. =
75. fields
76. .
77. DateTimeField
78. (auto_now_add
79. =
80. True
81. )
```

```
82. class
83. Service
84. (
85. models
86. .
87. Model
88. ):
89. id
90. =
91. fields
92. .
93. IntField
94. (pk
95. =
96. True
97. )
98. type
99. =
100. fields
101. .
102. IntEnumField
103. (ServiceType)
104. class
105. Provider
106. (
107. models
108. .
109. Model
110. ):
111. id
112. =
113. fields
114. .
115. IntField
116. (pk
117. =
118. True
119. )
120. name
121. =
122. fields
123. .
124. CharField
125. (max_length
126. =
127. 64
128. )
129. location
130. =
131. fields
132. .
133. CharField
134. (max_length
135. =
136. 64
137. )
138. created_at
139. =
140. fields
141. .
142. DateTimeField
143. (auto_now_add
144. =
145. True
146. )
147. availability
148. =
149. fields
```

```
150. .
151. ReverseRelation
152. [
153. "Availability"
154. ]
155. services
156. =
157. fields
158. .
159. ManyToManyField
160. (
161. "models.Service"
162. )
163. markup
164. =
165. fields
166. .
167. FloatField
168. (default
169. =
170. 1.1
171. )
172. class
173. Availability
174. (
175. models
176. .
177. Model
178. ):
179. id
180. =
181. fields
182. .
183. IntField
184. (pk
185. =
186. True
187. )
188. provider
189. =
190. fields
191. .
192. OneToOneField
193. (
194. "models.Provider"
195. , related_name
196. =
197. "availability"
198. )
199. max_distance
200. =
201. fields
202. .
203. IntField
204. (default
205. =
206. 10
207. )
208. time_start
209. =
210. fields
211. .
212. DateTimeField
213. ()
214. time_end
215. =
216. fields
217. .
```

```

218. DatetimeField
219. ()
220. min_hourly_price
221. =
222. fields
223. .
224. FloatField
225. (default
226. =
227. 0.0
228. )
229. We defined the following data models:
230.
231.     ◦ ServiceType
232.     ◦ is an enumeration class that defines different types of cleaning services. Each service type is associated with an
233.       integer value. Service types include FLOOR
234.     ◦ , WINDOW
235.     ◦ , LAUNDRY
236.     ◦ , IRON
237.     ◦ , and BATHROOM
238.     ◦ .
239.     ◦ User
240.     ◦ is a model representing the users of the cleaning service. It takes in the following fields:
241.     ◦
242.       ▪ id
243.     ◦
244.       ▪ : Auto-incrementing integer field serving as the primary key.
245.     ◦
246.       ▪ name
247.     ◦
248.       ▪ : Character field with a maximum length of 64 for the user's name.
249.     ◦
250.       ▪ address
251.     ◦
252.       ▪ : Character field with a maximum length of 100 for the user's address.
253.     ◦
254.       ▪ created_at
255.     ◦
256.       ▪ : DateTime field automatically set to the current timestamp when the user is created.
257.     ◦ Service
258.     ◦ is model representing types of cleaning services. It takes in the following fields:
259.     ◦
260.       ▪ id
261.     ◦
262.       ▪ : Auto-incrementing integer field serving as the primary key.
263.     ◦
264.       ▪ type

```

252.

- - : Enum field storing the ServiceType enumeration values.

253.

- Provider

254.

- is a model representing cleaning service providers. It considers the following fields:

255.

- - id

256.

- - : Auto-incrementing integer field serving as the primary key.

257.

- - name

258.

- - : Character field with a maximum length of 64 for the provider's name.

259.

- - location

260.

- - : Character field with a maximum length of 64 for the provider's location.

261.

- - created_at

262.

- - : DateTime field automatically set to the current timestamp when the provider is created.

263.

- - availability

264.

- - : Reverse relation to theAvailability

265.

- - model.

266.

- - services

267.

- - : Many-to-many relationship with theService

268.

- - model.

269.

- - markup

270.

- - : Float field representing the markup applied to service prices.

271.

- Availability

272.

- is a model representing the availability of a cleaning service provider. It considers the following fields:

273.

- - id

274.

- - : Auto-incrementing integer field serving as the primary key.

275.

- - provider

```

276.     ◦
277.         ▪ : One-to-one relationship to theProvider
278.     ◦
279.         ▪ model.
280.     ◦
281.         ▪ max_distance
282.     ◦
283.         ▪ : Integer field representing the maximum distance a provider can travel for services.
284.     ◦
285.         ▪ time_start
286.     ◦
287.         ▪ : DateTime field indicating the start time of availability.
288.     ◦
289.         ▪ time_end
290.     ◦
291.         ▪ : DateTime field indicating the end time of availability.
292.     ◦
293.         ▪ min_hourly_price
294.     ◦
295.         ▪ : 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:touchinit.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. .

```

```
29. geocoders
30. import
31. Nominatim
32. from
33. geopy
34. .
35. distance
36. import
37. geodesic
38. from
39. uagents
40. import
41. Context
42. ,
43. Model
44. ,
45. Protocol
46. from
47. .
48. models
49. import
50. Provider
51. ,
52. Availability
53. ,
54. User
55. PROTOCOL_NAME
56. =
57. "cleaning"
58. PROTOCOL_VERSION
59. =
60. "0.1.0"
61. class
62. ServiceRequest
63. (
64. Model
65. ):
66. user
67. :
68. str
69. location
70. :
71. str
72. time_start
73. :
74. datetime
75. duration
76. :
77. timedelta
78. services
79. :
80. List
81. [
82. int
83. ]
84. max_price
85. :
86. float
87. class
88. ServiceResponse
89. (
90. Model
91. ):
92. accept
93. :
94. bool
95. price
96. :
```



```

97. float
98. class
99. ServiceBooking
100. (
101. Model
102. ):
103. location
104. :
105. str
106. time_start
107. :
108. datetime
109. duration
110. :
111. timedelta
112. services
113. :
114. List
115. [
116. int
117. ]
118. price
119. :
120. float
121. class
122. BookingResponse
123. (
124. Model
125. ):
126. success
127. :
128. bool
129. cleaning_proto
130. =
131. Protocol
132. (name
133. =
134. PROTOCOL_NAME, version
135. =
136. PROTOCOL_VERSION)
137. We would then need to define anin_service_region
138. function, which determines whether a user's location is within the service area of a cleaning service provider. This
    function uses thegeopy
139. library for geolocation-related calculations and comparisons:
140. def
141. in_service_region
142. (
143. location
144. :
145. str
146. ,
147. availability
148. :
149. Availability
150. ,
151. provider
152. :
153. Provider
154. )
155. ->
156. bool
157. :
158. geocator
159. =
160. Nominatim
161. (user_agent
162. =
163. "micro_agents"

```

```
164. )
165. user_location
166. =
167. geolocator
168. .
169. geocode
170. (location)
171. cleaner_location
172. =
173. geolocator
174. .
175. geocode
176. (provider.location)
177. if
178. user_location
179. is
180. None
181. :
182. raise
183. RuntimeError
184. (
185. f
186. "user location
187. {
188. location
189. }
190. not found"
191. )
192. if
193. cleaner_location
194. is
195. None
196. :
197. raise
198. RuntimeError
199. (
200. f
201. "provider location
202. {
203. provider.location
204. }
205. not found"
206. )
207. cleaner_coordinates
208. =
209. (cleaner_location
210. .
211. latitude
212. ,
213. cleaner_location
214. .
215. longitude)
216. user_coordinates
217. =
218. (user_location
219. .
220. latitude
221. ,
222. user_location
223. .
224. longitude)
225. service_distance
226. =
227. geodesic
228. (user_coordinates, cleaner_coordinates).
229. miles
230. in_range
231. =
```

```

232. service_distance
233. <=
234. availability
235. .
236. max_distance
237. return
238. in_range
239. This function takes in the following parameters:
240.
    ◦ location
241.
    ◦ : The location of the user as a string (e.g., an address).
242.
    ◦ availability
243.
    ◦ : An instance of theAvailability
244.
    ◦ class that holds information about the provider's availability.
245.
    ◦ provider
246.
    ◦ : An instance of theProvider
247.
    ◦ class that holds information about the cleaning service provider.
248. The function starts by creating a geolocator object from theNominatim
249. class with a user agent stringmicro_agents
250. . Thegeolocator
251. 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
252. 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
253. function from thegeopy.distance
254. module. The calculated distance is stored in theservice_distance
255. variable. Finally, the function compares the calculatedservice_distance
256. 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 their_range
257. variable is set toTrue
258. . Otherwise, their_range
259. variable is set toFalse
260. . The function returns the value ofin_range
261. , which indicates whether the user's location is within the service area of the provider.
262. We then need to define an event handler using the@cleaning_proto.on_message
263. 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
264. messages and generatesServiceResponse
265. messages as responses:
266. @cleaning_proto
267. .
268. on_message
269. (model
270. =
271. ServiceRequest, replies
272. =
273. ServiceResponse)
274. async
275. def
276. handle_query_request
277. (
278. ctx
279. :
280. Context
281. ,
282. sender
283. :
284. str
285. ,

```

```
286. msg
287. :
288. ServiceRequest):
289. provider
290. =
291. await
292. Provider
293. .
294. filter
295. (name
296. =
297. ctx.name).
298. first
299. ()
300. availability
301. =
302. await
303. Availability
304. .
305. get
306. (provider
307. =
308. provider)
309. services
310. =
311. [
312. int
313. (service.type)
314. for
315. service
316. in
317. await
318. provider
319. .
320. services]
321. markup
322. =
323. provider
324. .
325. markup
326. user
327. ,
328. _
329. =
330. await
331. User
332. .
333. get_or_create
334. (name
335. =
336. msg.user, address
337. =
338. sender)
339. msg_duration_hours
340. :
341. float
342. =
343. msg
344. .
345. duration
346. .
347. total_seconds
348. ()
349. /
350. 3600
351. ctx
352. .
353. logger
```

```
354. .
355. info
356. (
357. f
358. "Received service request from user `
359. {
360. user.name
361. }
362. ``
363. )
364. if
365. (
366. set
367. (msg.services)
368. <=
369. set
370. (services)
371. and
372. in_service_region
373. (msg.location, availability, provider)
374. and
375. availability
376. .
377. time_start
378. <=
379. msg
380. .
381. time_start
382. and
383. availability
384. .
385. time_end

386.      =

387. msg
388. .
389. time_start
390. +
391. msg
392. .
393. duration
394. and
395. availability
396. .
397. min_hourly_price
398. *
399. msg_duration_hours
400. <
401. msg
402. .
403. max_price
404. )
405. :
406. accept
407. =
408. True
409. price
410. =
411. markup
412. *
413. availability
414. .
415. min_hourly_price
416. *
417. msg_duration_hours
418. ctx
419. .
420. logger
```

```

421. .
422. info
423. (
424. f
425. "I am available! Proposing price:
426. {
427. price
428. }
429. ."
430. )
431. else
432. :
433. accept
434. =
435. False
436. price
437. =
438. 0
439. ctx
440. .
441. logger
442. .
443. info
444. (
445. "I am not available. Declining request."
446. )
447. await
448. ctx
449. .
450. send
451. (sender,
452. ServiceResponse
453. (accept
454. =
455. accept, price
456. =
457. price))
458. We defined a@cleaning_proto.on_message()
459. decorator. Within the decorator, we defined thehandle_query_request()
460. function which is the actual event handler that gets executed when aServiceRequest
461. message is received. It takes the following arguments:ctx
462. ,sender
463. , andmsg
464. .
465. The handler first retrieves the provider's information, availability, services, and markup using the context's information.
    It uses theUser.get_or_create
466. 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:
467.
    ◦ Whether the requested services are within the provider's available services.
468.
    ◦ Whether the user's location is within the provider's service area using their_service_region
469.
    ◦ function.
470.
    ◦ Whether the requested time is within the provider's availability.
471.
    ◦ Whether the calculated price based on the minimum hourly price and markup is within the user's maximum price.
472. If all conditions are met, the handler setsaccept
473. toTrue
474. and calculates the proposedprice
475. based on the markup and availability's minimum hourly price. If any condition is not met, the handler setsaccept
476. toFalse
477. and theprice
478. to0
479. . Finally, the handler sends aServiceResponse
480. message back to the sender with the calculatedaccept

```

```
481. flag and price
482. . The handler logs relevant information using the ctx.logger.info()
483. method.
484. We then define another event handler, handle_book_request()
485. function, using the @cleaning_proto.on_message()
486. 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
    a ServiceBooking
487. message is received and generates a BookingResponse
488. messages as response:
489. @cleaning_proto
490. .
491. on_message
492. (model
493. =
494. ServiceBooking, replies
495. =
496. BookingResponse)
497. async
498. def
499. handle_book_request
500. (
501. ctx
502. :
503. Context
504. ,
505. sender
506. :
507. str
508. ,
509. msg
510. :
511. ServiceBooking):
512. provider
513. =
514. await
515. Provider
516. .
517. filter
518. (name
519. =
520. ctx.name).
521. first
522. ()
523. availability
524. =
525. await
526. Availability
527. .
528. get
529. (provider
530. =
531. provider)
532. services
533. =
534. [
535. int
536. (service.type)
537. for
538. service
539. in
540. await
541. provider
542. .
543. services]
544. user
545. =
546. await
```

```
547. User
548. .
549. get
550. (address
551. =
552. sender)
553. msg_duration_hours
554. :
555. float
556. =
557. msg
558. .
559. duration
560. .
561. total_seconds
562. ()
563. /
564. 3600
565. ctx
566. .
567. logger
568. .
569. info
570. (
571. f
572. "Received booking request from user `
573. {
574. user.name
575. }
576. `"
577. )
578. success
579. =
580. (
581. set
582. (msg.services)
583. <=
584. set
585. (services)
586. and
587. availability
588. .
589. time_start
590. <=
591. msg
592. .
593. time_start
594. and
595. availability
596. .
597. time_end

598.      =

599. msg
600. .
601. time_start
602. +
603. msg
604. .
605. duration
606. and
607. msg
608. .
609. price
610. <=
611. availability
612. .
613. min_hourly_price
```



```

614. *
615. msg_duration_hours
616. )
617. if
618. success
619. :
620. availability
621. .
622. time_start
623. =
624. msg
625. .
626. time_start
627. +
628. msg
629. .
630. duration
631. await
632. availability
633. .
634. save
635. ()
636. ctx
637. .
638. logger
639. .
640. info
641. (
642. "Accepted task and updated availability."
643. )

```

644. **send the response**

```

645. await
646. ctx
647. .
648. send
649. (sender,
650. BookingResponse
651. (success
652. =
653. success))
654. We defined the @cleaning_proto.on_message()
655. decorator which is triggered when aServiceBooking
656. message is received. The event handler handle_book_request()
657. function gets executed when aServiceBooking
658. message is received. It takes the following parameters: ctx
659. , sender
660. , and msg
661. . 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:
662.
    ◦ Whether the requested services are within the provider's available services.
663.
    ◦ Whether the requested time is within the provider's availability.
664.
    ◦ Whether the requested price is within the acceptable range based on the availability's minimum hourly price and
        duration.
665. 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 the ctx.logger.info()
666. method to log that the task has been accepted and availability has been updated. Regardless of success, the handler
    sends a BookingResponse
667. message back to the sender with a flag indicating whether the booking request was successful or not.
668. 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 our cleaner 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 our cleaner
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 specifcname
7. ,port
8. ,seed
9. , andendpoint
10. :
11. from
12. datetime
13. import

```
14. datetime
15. from
16. pytz
17. import
18. utc
19. from
20. tortoise
21. import
22. Tortoise
23. from
24. protocols
25. .
26. cleaning
27. import
28. cleaning_proto
29. from
30. protocols
31. .
32. cleaning
33. .
34. models
35. import
36. Availability
37. ,
38. Provider
39. ,
40. Service
41. ,
42. ServiceType
43. from
44. uagents
45. import
46. Agent
47. ,
48. Context
49. from
50. uagents
51. .
52. setup
53. import
54. fund_agent_if_low
55. cleaner
56. =
57. Agent
58. (
59. name
60. =
61. "cleaner"
62. ,
63. port
64. =
65. 8001
66. ,
67. seed
68. =
69. "cleaner secret phrase"
70. ,
71. endpoint
72. =
73. {
74. "http://127.0.0.1:8001/submit"
75. : {},
76. },
77. )
78. fund_agent_if_low
79. (cleaner.wallet.
80. address
81. ())
```

82. **build the cleaning service agent from the cleaning protocol**

```
83. cleaner
84. .
85. include
86. (cleaning_proto)
87. Thecleaner
88. agent includes the previously defined cleaning protocol (cleaning_proto
89. ) using theinclude()
90. 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.
91. We then define the behaviors of ourcleaner
92. agent:
93. @cleaner
94. .
95. on_event
96. (
97. "startup"
98. )
99. async
100. def
101. startup
102. (
103. _ctx
104. :
105. Context):
106. await
107. Tortoise
108. .
109. init
110. (
111. db_url
112. =
113. "sqlite:///db.sqlite3"
114. , modules
115. =
116. {
117. "models"
118. : [
119. "protocols.cleaning.models"
120. ]}
121. )
122. await
123. Tortoise
124. .
125. generate_schemas
126. ()
127. provider
128. =
129. await
130. Provider
131. .
132. create
133. (name
134. =
135. cleaner.name, location
136. =
137. "London Kings Cross"
138. )
139. floor
140. =
141. await
142. Service
```

```
143. .
144. create
145. (type
146. =
147. ServiceType.FLOOR)
148. window
149. =
150. await
151. Service
152. .
153. create
154. (type
155. =
156. ServiceType.WINDOW)
157. laundry
158. =
159. await
160. Service
161. .
162. create
163. (type
164. =
165. ServiceType.LAUNDRY)
166. await
167. provider
168. .
169. services
170. .
171. add
172. (floor)
173. await
174. provider
175. .
176. services
177. .
178. add
179. (window)
180. await
181. provider
182. .
183. services
184. .
185. add
186. (laundry)
187. await
188. Availability
189. .
190. create
191. (
192. provider
193. =
194. provider,
195. time_start
196. =
197. utc.
198. localize
199. (datetime.
200. fromisoformat
201. (
202. "2022-01-31 00:00:00"
203. )),
204. time_end
205. =
206. utc.
207. localize
208. (datetime.
209. fromisoformat
210. (
```

```

211. "2023-05-01 00:00:00"
212. )),
213. max_distance
214. =
215. 10
216. ,
217. min_hourly_price
218. =
219. 5
220. ,
221. )
222. @cleaner
223. .
224. on_event
225. (
226. "shutdown"
227. )
228. async
229. def
230. shutdown
231. (
232. _ctx
233. :
234. Context):
235. await
236. Tortoise
237. .
238. close_connections
239. ()
240. if
241. name
242. ==
243. "main"
244. :
245. cleaner
246. .
247. run
248. ()
249. Thestartup()
250. event handler function is decorated with the.on_event()
251. decorator. This handler is executed when thecleaner
252. 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
253. is set tosqlite://db.sqlite3
254. , indicating the use of an SQLite database nameddb.sqlite3
255. .modules
256. specifies the module containing the data models related to the cleaning protocol. We then define
    theTortoise.generate_schemas()
257. 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
258. andService
259. models and populates them with data.
260. AProvider
261. instance is created with thecleaner
262. agent's name ("cleaner") and location ("London Kings Cross").
263. ThreeService
264. instances are created with different service types:FLOOR
265. ,WINDOW
266. , andLAUNDRY
267. . The createdService
268. instances are associated with theProvider
269. instance using theprovider.services.add()
270. method. This establishes a relationship between providers and the services they offer. We proceed by defining
    anAvailability
271. 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 our user 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

```

```
69. =
70. 8000
71. ,
72. seed
73. =
74. "cleaning user recovery phrase"
75. ,
76. endpoint
77. =
78. {
79. "http://127.0.0.1:8000/submit"
80. : {},
81. },
82. )
83. fund_agent_if_low
84. (user.wallet.
85. address
86. ())
87. Let's define aServiceRequest
88. 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
89. to apply a discount or markdown to the service:
90. request
91. =
92. ServiceRequest
93. (
94. user
95. =
96. user.name,
97. location
98. =
99. "London Kings Cross"
100. ,
101. time_start
102. =
103. utc.
104. localize
105. (datetime.
106. fromisoformat
107. (
108. "2023-04-10 16:00:00"
109. )),
110. duration
111. =
112. timedelta
113. (hours
114. =
115. 4
116. ),
117. services
118. =
119. [ServiceType.WINDOW, ServiceType.LAUNDRY],
120. max_price
121. =
122. 60
123. ,
124. )
125. MARKDOWN
126. =
127. 0.8
128. We then define an event handler to handle service requests:
129. @user
130. .
131. on_interval
132. (period
133. =
134. 3.0
135. , messages
```

```

136. =
137. ServiceRequest)
138. async
139. def
140. interval
141. (
142. ctx
143. :
144. Context):
145. ctx
146. .
147. storage
148. .
149. set
150. (
151. "markdown"
152. , MARKDOWN)
153. completed
154. =
155. ctx
156. .
157. storage
158. .
159. get
160. (
161. "completed"
162. )
163. if
164. not
165. completed
166. :
167. ctx
168. .
169. logger
170. .
171. info
172. (
173. f
174. "Requesting cleaning service:
175. {
176. request
177. }
178. "
179. )
180. await
181. ctx
182. .
183. send
184. (CLEANER_ADDRESS, request)
185. Theinterval()
186. function is decorated with the.on_interval()
187. decorator indicating that is is executed at regular intervals specified by the period parameter to sendServiceRequest
188. messages. Within the function, thectx.storage.set()
189. method is called to store theMARKDOWN
190. constant in the context's storage under the key "markdown". The function first checks the value ofcompleted
191. which is retrieved from the context's storage usingctx.storage.get()
192. 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()
193. method indicating that a cleaning service request is being made. The{request}
194. placeholder in the log message is filled with the details of the request object previously created. Then,ctx.send()
195. method is called to send the cleaning service request to the specified address (CLEANER_ADDRESS
196. ) alongside with the request object.
197. We then would need to define a function for handling responses to service queries and generating service bookings
    based on the received response:
198. @user
199. .
200. on_message
201. (ServiceResponse, replies

```

```
202. =
203. ServiceBooking)
204. async
205. def
206. handle_query_response
207. (
208. ctx
209. :
210. Context
211. ,
212. sender
213. :
214. str
215. ,
216. msg
217. :
218. ServiceResponse):
219. markdown
220. =
221. ctx
222. .
223. storage
224. .
225. get
226. (
227. "markdown"
228. )
229. if
230. msg
231. .
232. accept
233. :
234. ctx
235. .
236. logger
237. .
238. info
239. (
240. "Cleaner is available, attempting to book now"
241. )
242. booking
243. =
244. ServiceBooking
245. (
246. location
247. =
248. request.location,
249. time_start
250. =
251. request.time_start,
252. duration
253. =
254. request.duration,
255. services
256. =
257. request.services,
258. price
259. =
260. markdown
261. *
262. msg.price,
263. )
264. await
265. ctx
266. .
267. send
268. (sender, booking)
269. else
```

```

270. :
271. ctx
272. .
273. logger
274. .
275. info
276. (
277. "Cleaner is not available - nothing more to do"
278. )
279. ctx
280. .
281. storage
282. .
283. set
284. (
285. "completed"
286. ,
287. True
288. )
289. Thehandle_query_response()
290. function is decorated with.on_message()
291. and handles messages of typeServiceResponse
292. . It also specifies that the function replies with instances ofServiceBooking
293. data model.
294. The function first retrieves the markdown value from the context's storage usingctx.storage.get()
295. method. If the receivedServiceResponse
296. 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
297. 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
298. object is then sent to the sender's address usingctx.send()
299. method.
300. 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
301. , indicating that the service request process has been completed.
302. We finally need to define a function for handlingBookingResponse
303. messages:
304. @user
305. .
306. on_message
307. (BookingResponse, replies
308. =
309. set
310. ())
311. async
312. def
313. handle_book_response
314. (
315. ctx
316. :
317. Context
318. ,
319. _sender
320. :
321. str
322. ,
323. msg
324. :
325. BookingResponse):
326. if
327. msg
328. .
329. success
330. :
331. ctx
332. .
333. logger
334. .

```

```

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](#)