

How to book a table at a restaurant using agents

Introduction

In this guide, we want to show how to set up the code to create a restaurant booking service with two AI Agents: a restaurant with tables available, and a user requesting table availability.

We also want to define 2 specific protocols, one for table querying (i.e., Table querying protocol) and one for table booking (i.e., Table booking protocol). Then we will need to define two agents scripts, restaurant and user, which will make use of the protocols to query and book a table.

Walk-through

1. First of all, we need to navigate towards the directory you created for your project and create a folder for this task: `mkdir booking_demo`
2. `.`
3. Inside this folder we will create another folder for our protocols
4. `:mkdir protocols`
5. `.`
6. After having defined our protocols, we will create 2 scripts, one for our restaurant
7. and the other one for user
8. agents. These agents will make use of the protocols to query, check, confirm and book an available table at the restaurant.

We can start by writing the code for our 2 protocols.

Protocols

Table querying protocol

Let's start by defining the protocol for querying availability of tables at the restaurant:

1. First of all, we need to navigate towards the protocols folder: `cd protocols`
2. Let's then create a Python script and name it by running: `touch query.py`
3. In the text editor application, we proceed and define the table querying protocol
4. `.`
5. We now need to import necessary classes and define the message data models
6. `.` Then, create an instance of the Protocol
7. class and name it `query_proto`
8. `:`
9. `from`
10. `typing`
11. `import`
12. `List`
13. `from`
14. `uagents`
15. `import`
16. `Context`
17. `,`
18. `Model`
19. `,`
20. `Protocol`
21. `class`
22. `TableStatus`
23. `(`
24. `Model`
25. `):`
26. `seats`
27. `:`
28. `int`
29. `time_start`
30. `:`
31. `int`
32. `time_end`
33. `:`
34. `int`

```

35. class
36. QueryTableRequest
37. (
38. Model
39. ):
40. guests
41. :
42. int
43. time_start
44. :
45. int
46. duration
47. :
48. int
49. class
50. QueryTableResponse
51. (
52. Model
53. ):
54. tables
55. :
56. List
57. [
58. int
59. ]
60. class
61. GetTotalQueries
62. (
63. Model
64. ):
65. pass
66. class
67. TotalQueries
68. (
69. Model
70. ):
71. total_queries
72. :
73. int
74. query_proto
75. =
76. Protocol
77. ()
78.
79.     ◦ TableStatus
80.     ◦ : this represents the status of a table and includes the attributes number of seats, start time, and end time.
81.     ◦ QueryTableRequest
82.     ◦ : this is used for querying table availability. It includes information about the number of guests, start time, and
83.       duration of the table request.
84.     ◦ QueryTableResponse
85.     ◦ : this contains the response to the query table availability. It includes a list of table numbers that are available
86.       based on query parameters.
87.     ◦ GetTotalQueries
88.     ◦ : this is used to request the total number of queries made to the system.
89.     ◦ TotalQueries
90.     ◦ : this contains the response to the total queries request, including the count of total queries made to the system.
91. Let's then define the message handlers for the query_proto
92. protocol:
93. @query_proto

```

```
91. .
92. on_message
93. (model
94. =
95. QueryTableRequest, replies
96. =
97. QueryTableResponse)
98. async
99. def
100. handle_query_request
101. (
102. ctx
103. :
104. Context
105. ,
106. sender
107. :
108. str
109. ,
110. msg
111. :
112. QueryTableRequest):
113. tables
114. =
115. {
116. int
117. (num):
118. TableStatus
119. (
120. **
121. status)
122. for
123. (
124. num
125. ,
126. status
127. ,
128. )
129. in
130. ctx
131. .
132. storage
133. .
134. _data
135. .
136. items
137. ()
138. if
139. isinstance
140. (num,
141. int
142. )
143. }
144. available_tables
145. =
146. []
147. for
148. number
149. ,
150. status
151. in
152. tables
153. .
154. items
155. ():
156. if
157. (
158. status
```

```
159. .
160. seats

161.      =

162. msg
163. .
164. guests
165. and
166. status
167. .
168. time_start
169. <=
170. msg
171. .
172. time_start
173. and
174. status
175. .
176. time_end

177.      =

178. msg
179. .
180. time_start
181. +
182. msg
183. .
184. duration
185. )
186. :
187. available_tables
188. .
189. append
190. (
191. int
192. (number))
193. ctx
194. .
195. logger
196. .
197. info
198. (
199. f
200. "Query:
201. {
202. msg
203. }
204. . Available tables:
205. {
206. available_tables
207. }
208. ."
209. )
210. await
211. ctx
212. .
213. send
214. (sender,
215. QueryTableResponse
216. (tables
217. =
218. available_tables))
219. total_queries
220. =
221. int
222. (ctx.storage.
223. get
```

```

224. (
225. "total_queries"
226. )
227. or
228. 0
229. )
230. ctx
231. .
232. storage
233. .
234. set
235. (
236. "total_queries"
237. , total_queries
238. +
239. 1
240. )
241. @query_proto
242. .
243. on_query
244. (model
245. =
246. GetTotalQueries, replies
247. =
248. TotalQueries)
249. async
250. def
251. handle_get_total_queries
252. (
253. ctx
254. :
255. Context
256. ,
257. sender
258. :
259. str
260. ,
261. _msg
262. :
263. GetTotalQueries):
264. total_queries
265. =
266. int
267. (ctx.storage.
268. get
269. (
270. "total_queries"
271. )
272. or
273. 0
274. )
275. await
276. ctx
277. .
278. send
279. (sender,
280. TotalQueries
281. (total_queries
282. =
283. total_queries))
284.
    ◦ handle_query_request()
285.
    ◦ : this message handler function is defined using the.on_message()
286.
    ◦ decorator. It handles theQueryTableRequest
287.
    ◦ messages and replies with aQueryTableResponse

```

- 288.
- message. The handler processes the table availability query based on the provided parameters, checks the table statuses stored in the agent's storage, and sends the available table numbers as a response to the querying agent. Additionally, the handler tracks the total number of queries made and increments the count in storage.
- 289.
- `handle_get_total_queries()`
- 290.
- : this message handler function is defined using the `on_query()`
- 291.
- decorator. It handles the `GetTotalQueries`
- 292.
- query and replies with a `TotalQueries`
- 293.
- message containing the total number of queries made to the system. The handler retrieves the total query count from the agent's storage and responds with the count.
294. Save the script.

The overall script should look as follows:

```
query.py from typing import List
```

```
from uagents import Context , Model , Protocol
```

```
class
```

```
TableStatus ( Model ): seats :
```

```
int time_start :
```

```
int time_end :
```

```
int
```

```
class
```

```
QueryTableRequest ( Model ): guests :
```

```
int time_start :
```

```
int duration :
```

```
int
```

```
class
```

```
QueryTableResponse ( Model ): tables : List [ int ]
```

```
class
```

```
GetTotalQueries ( Model ): pass
```

```
class
```

```
TotalQueries ( Model ): total_queries :
```

```
int query_proto =
```

```
Protocol ()
```

```
@query_proto . on_message (model = QueryTableRequest, replies = QueryTableResponse) async
```

```
def
```

```
handle_query_request ( ctx : Context ,
```

```
sender :
```

```
str ,
```

```
msg : QueryTableRequest): tables =
```

```
{ int (num):
```

```

TableStatus ( ** status) for ( num , status , ) in ctx . storage . _data . items () if
isinstance (num,int ) } available_tables = [] for number , status in tables . items (): if ( status . seats

    = msg . guests and status . time_start <= msg . time_start and status . time_end = msg . time_start + msg .
    duration ) : available_tables . append ( int (number)) ctx . logger . info ( f "Query: { msg } . Available tables: {
    available_tables } ." ) await ctx . send (sender, QueryTableResponse (tables = available_tables)) total_queries =

int (ctx.storage. get ( "total_queries" ) or

0 ) ctx . storage . set ( "total_queries" , total_queries +

1 )

@query_proto . on_query (model = GetTotalQueries, replies = TotalQueries) async
def

handle_get_total_queries ( ctx : Context ,

sender :

str ,

_msg : GetTotalQueries): total_queries =

int (ctx.storage. get ( "total_queries" ) or

0 ) await ctx . send (sender, TotalQueries (total_queries = total_queries))

```

Table booking protocol

We can now proceed by writing the booking protocol script for booking the table at the restaurant .

1. First of all, navigate towards the protocols folder:cd protocols
2. In here, let's create a Python script and name it by running:touch book.py
3. In the text editor application, we can proceed to define the table booking protocol
4. . We need to import the necessary classes and define the message data models
5. . In this case, the booking protocol consists of two message models:BookTableRequest
6. andBookTableResponse
7. . Then create an instance of theProtocol
8. class and name itbook_proto
9. :
10. from
11. uagents
12. import
13. Context
14. ,
15. Model
16. ,
17. Protocol
18. from
19. .
20. query
21. import
22. TableStatus
23. class
24. BookTableRequest
25. (
26. Model
27.):
28. table_number
29. :
30. int
31. time_start
32. :
33. int
34. duration
35. :
36. int

```

37. class
38. BookTableResponse
39. (
40. Model
41. ):
42. success
43. :
44. bool
45. book_proto
46. =
47. Protocol
48. ()
49.
    ◦ BookTableRequest
50.
    ◦ : this represents the request to book a table. It includes attributes:table_number
51.
    ◦ to be booked,time_start
52.
    ◦ of the booking, and theduration
53.
    ◦ of the booking.
54.
    ◦ BookTableResponse
55.
    ◦ : this contains the response to the table booking request. It includes a boolean attributesuccess
56.
    ◦ , indicating whether the booking was successful or not.
57. Let's now define a message handler function:
58. @book_proto
59. .
60. on_message
61. (model
62. =
63. BookTableRequest, replies
64. =
65. BookTableResponse)
66. async
67. def
68. handle_book_request
69. (
70. ctx
71. :
72. Context
73. ,
74. sender
75. :
76. str
77. ,
78. msg
79. :
80. BookTableRequest):
81. tables
82. =
83. {
84. int
85. (num):
86. TableStatus
87. (
88. **
89. status)
90. for
91. (
92. num
93. ,
94. status
95. ,
96. )

```



```
97. in
98. ctx
99. .
100. storage
101. .
102. _data
103. .
104. items
105. ()
106. if
107. isinstance
108. (num,
109. int
110. )
111. }
112. table
113. =
114. tables
115. [
116. msg
117. .
118. table_number
119. ]
120. if
121. (
122. table
123. .
124. time_start
125. <=
126. msg
127. .
128. time_start
129. and
130. table
131. .
132. time_end

133.      =

134. msg
135. .
136. time_start
137. +
138. msg
139. .
140. duration
141. )
142. :
143. success
144. =
145. True
146. table
147. .
148. time_start
149. =
150. msg
151. .
152. time_start
153. +
154. msg
155. .
156. duration
157. ctx
158. .
159. storage
160. .
161. set
162. (msg.table_number, table.
163. dict
```

```
164. ())
165. else
166. :
167. success
168. =
169. False
```

170. **send the response**

```
171. await
172. ctx
173. .
174. send
175. (sender,
176. BookTableResponse
177. (success
178. =
179. success))
180. Thehandle_book_request()
181. handler first retrieves table statuses from the agent's storage and converts them into a dictionary with integer keys
    (table numbers) andTableStatus
182. values. TheTableStatus
183. class is imported from thequery
184. module. Next, the handler gets the table associated with the requestedtable_number
185. from thetables
186. dictionary. The handler checks if the requestedtime_start
187. falls within the availability period of the table. If the table is available for the requested booking duration, the handler
    setssuccess
188. toTrue
189. , updates the table'stime_start
190. to reflect the end of the booking, and saves the updated table information in the agent's storage usingctx.storage.set()
191. . If the table is not available for the requested booking, the handler setssuccess
192. toFalse
193. .
194. The handler sends aBookTableResponse
195. message back to the sender with thesuccess
196. status of the booking using awaitctx.send()
197. method.
198. Save the script.
```

The overall script should look as follows:

```
book.py from uagents import Context , Model , Protocol from
```

```
. query import TableStatus
```

```
class
```

```
BookTableRequest ( Model ): table_number :
```

```
int time_start :
```

```
int duration :
```

```
int
```

```
class
```

```
BookTableResponse ( Model ): success :
```

```
bool
```

book_proto

```
Protocol () @book_proto . on_message (model = BookTableRequest, replies = BookTableResponse) async
```

```
def
```

```

handle_book_request ( ctx : Context ,
sender :
str ,
msg : BookTableRequest): tables =
{ int (num):
TableStatus ( ** status) for ( num , status , ) in ctx . storage . _data . items () if
isinstance (num, int ) } table = tables [ msg . table_number ] if ( table . time_start <= msg . time_start and table . time_end
= msg . time_start + msg . duration ) : success =
True table . time_start = msg . time_start + msg . duration ctx . storage . set (msg.table_number, table. dict ()) else :
success =
False

```

send the response

```

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

```

Restaurant Agent

We are now ready to define our restaurant agent.

1. Let's now create a Python script in booking_demo
2. folder, and name it by running: touch restaurant.py
3. We then need to import the necessary classes from the agents
4. library and the two protocols we previously defined:
5. from
6. uagents
7. import
8. Agent
9. from
10. uagents
11. .
12. setup
13. import
14. fund_agent_if_low
15. restaurant
16. =
17. Agent
18. (
19. name
20. =
21. "restaurant"
22. ,
23. port
24. =
25. 8001
26. ,
27. seed
28. =
29. "restaurant secret phrase"
30. ,
31. endpoint
32. =
33. [
34. "http://127.0.0.1:8001/submit"
35.],
36.)
37. fund_agent_if_low
38. (restaurant.wallet.
39. address
40. ())

```
41. Let's build therestaurant
42. agent from above protocols and set the table availability information:
43. from
44. protocols
45. .
46. book
47. import
48. book_proto
49. from
50. protocols
51. .
52. query
53. import
54. query_proto
55. ,
56. TableStatus
```

57. **build the restaurant agent from stock protocols**

```
58. restaurant
59. .
60. include
61. (query_proto)
62. restaurant
63. .
64. include
65. (book_proto)
66. TABLES
67. =
68. {
69. 1
70. :
71. TableStatus
72. (seats
73. =
74. 2
75. , time_start
76. =
77. 16
78. , time_end
79. =
80. 22
81. ),
82. 2
83. :
84. TableStatus
85. (seats
86. =
87. 4
88. , time_start
89. =
90. 19
91. , time_end
92. =
93. 21
94. ),
95. 3
96. :
97. TableStatus
98. (seats
99. =
100. 4
101. , time_start
102. =
103. 17
104. , time_end
105. =
```

```

106. 19
107. ),
108. }
109. We would then need to store the TABLES
110. information in the restaurant agent and run it:
111. for
112. (number
113. ,
114. status)
115. in
116. TABLES
117. .
118. items
119. ():
120. restaurant
121. .
122. _storage
123. .
124. set
125. (number, status.
126. dict
127. ())
128. if
129. name
130. ==
131. "main"
132. :
133. restaurant
134. .
135. run
136. ()
137. Save the script.

```

The restaurant agent is now online and ready to receive messages .

The overall script should look as follows:

```

restaurant.py from uagents import Agent , Context from uagents . setup import fund_agent_if_low from protocols . book
import book_proto from protocols . query import query_proto , TableStatus

```

restaurant

```

Agent ( name = "restaurant" , port = 8001 , seed = "restaurant secret phrase" , endpoint = [ "http://127.0.0.1:8001/submit" ], )

```

```

fund_agent_if_low (restaurant.wallet. address ())

```

build the restaurant agent from stock protocols

```

restaurant . include (query_proto) restaurant . include (book_proto) TABLES =

```

```

{ 1 :

```

```

TableStatus (seats = 2 , time_start = 16 , time_end = 22 ), 2 :

```

```

TableStatus (seats = 4 , time_start = 19 , time_end = 21 ), 3 :

```

```

TableStatus (seats = 4 , time_start = 17 , time_end = 19 ), }

```

set the table availability information in the restaurant protocols

```

for (number , status) in TABLES . items (): restaurant . _storage . set (number, status. dict ())

```

```

if

```

name

==

"main" : restaurant . run ()

User uAgent

We can finally define the script for our user agent querying and booking a table at the restaurant.

```
1. First of all, let's create a Python script in booking_demo
2. folder, and name it: touch user.py
3. We then need to import necessary classes from the uagents
4. library and the two protocols defined above. We also need the restaurant
5. agent's address to be able to communicate with it:
6. from
7. uagents
8. import
9. Agent
10. ,
11. Context
12. from
13. uagents
14. .
15. setup
16. import
17. fund_agent_if_low
18. from
19. protocols
20. .
21. book
22. import
23. BookTableRequest
24. ,
25. BookTableResponse
26. from
27. protocols
28. .
29. query
30. import
31. (
32. QueryTableRequest
33. ,
34. QueryTableResponse
35. ,
36. )
37. RESTAURANT_ADDRESS
38. =
39. "agent1qw50wcs4nd723ya9j8mwzglhns2kzzhh0et0yl34vr75hualsyqvqdzl990"
40. user
41. =
42. Agent
43. (
44. name
45. =
46. "user"
47. ,
48. port
49. =
50. 8000
51. ,
52. seed
53. =
54. "user secret phrase"
55. ,
56. endpoint
57. =
58. [
59. "http://127.0.0.1:8000/submit"
```

```

60. ],
61. )
62. fund_agent_if_low
63. (user.wallet.
64. address
65. ())
66. Let's then create the table query to generate theQueryTableRequest
67. using therestaurant
68. address. If the request has not been completed before, we send the request to the restaurant agent. Then create
   aninterval()
69. function which performs a table query request on a defined period to therestaurant
70. agent, to query the availability of a table given thetable_query
71. parameters:
72. table_query
73. =
74. QueryTableRequest
75. (
76. guests
77. =
78. 3
79. ,
80. time_start
81. =
82. 19
83. ,
84. duration
85. =
86. 2
87. ,
88. )
89. @user
90. .
91. on_interval
92. (period
93. =
94. 3.0
95. , messages
96. =
97. QueryTableRequest)
98. async
99. def
100. interval
101. (
102. ctx
103. :
104. Context):
105. completed
106. =
107. ctx
108. .
109. storage
110. .
111. get
112. (
113. "completed"
114. )
115. if
116. not
117. completed
118. :
119. await
120. ctx
121. .
122. send
123. (RESTAURANT_ADDRESS, table_query)
124. Define the message handler function for incomingQueryTableResponse
125. messages from therestaurant
126. agent:

```

```
127. @user
128. .
129. on_message
130. (QueryTableResponse, replies
131. =
132. {BookTableRequest})
133. async
134. def
135. handle_query_response
136. (
137. ctx
138. :
139. Context
140. ,
141. sender
142. :
143. str
144. ,
145. msg
146. :
147. QueryTableResponse):
148. if
149. len
150. (msg.tables)
151.
152. 0
153. :
154. ctx
155. .
156. logger
157. .
158. info
159. (
160. "There is a free table, attempting to book one now"
161. )
162. table_number
163. =
164. msg
165. .
166. tables
167. [
168. 0
169. ]
170. request
171. =
172. BookTableRequest
173. (
174. table_number
175. =
176. table_number,
177. time_start
178. =
179. table_query.time_start,
180. duration
181. =
182. table_query.duration,
183. )
184. await
185. ctx
186. .
187. send
188. (sender, request)
189. else
190. :
191. ctx
192. .
193. logger
194. .
```



```
195. info
196. (
197. "No free tables - nothing more to do"
198. )
199. ctx
200. .
201. storage
202. .
203. set
204. (
205. "completed"
206. ,
207. True
208. )
209. This function activates when a message is received back from therestaurant
210. agent.handle_query_response()
211. function will evaluate if there is a table available, and if so, it responds with aBookTableRequest
212. to complete the reservation.
213. Let's then define a function which will handle messages from therestaurant
214. agent on whether the reservation was successful or not:
215. @user
216. .
217. on_message
218. (BookTableResponse, replies
219. =
220. set
221. ())
222. async
223. def
224. handle_book_response
225. (
226. ctx
227. :
228. Context
229. ,
230. _sender
231. :
232. str
233. ,
234. msg
235. :
236. BookTableResponse):
237. if
238. msg
239. .
240. success
241. :
242. ctx
243. .
244. logger
245. .
246. info
247. (
248. "Table reservation was successful"
249. )
250. else
251. :
252. ctx
253. .
254. logger
255. .
256. info
257. (
258. "Table reservation was UNSUCCESSFUL"
259. )
260. ctx
261. .
262. storage
```

```

263. .
264. set
265. (
266. "completed"
267. ,
268. True
269. )
270. if
271. name
272. ==
273. "main"
274. :
275. user
276. .
277. run
278. ()
279. Save the script.

```

The overall script should look as follows:

```

user.py from protocols . book import BookTableRequest , BookTableResponse from protocols . query import (
QueryTableRequest , QueryTableResponse , ) from uagents import Agent , Context from uagents . setup import
fund_agent_if_low

```

RESTAURANT_ADDRESS

```
"agent1qw50wcs4nd723ya9j8mwzglhs2kzzhh0et0yl34vr75hualsyqvqdzl990"
```

user

```
Agent ( name = "user" , port = 8000 , seed = "user secret phrase" , endpoint = [ "http://127.0.0.1:8000/submit" ], )
```

```
fund_agent_if_low (user.wallet. address ())
```

table_query

```
QueryTableRequest ( guests = 3 , time_start = 19 , duration = 2 , )
```

This on_interval agent function performs a request on a defined period

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

```
def
```

```
interval ( ctx : Context): completed = ctx . storage . get ( "completed" )
```

```
if
```

```
not completed : await ctx . send (RESTAURANT_ADDRESS, table_query)
```

```
@user . on_message (QueryTableResponse, replies = {BookTableRequest}) async
```

```
def
```

```
handle_query_response ( ctx : Context ,
```

```
sender :
```

```
str ,
```

```
msg : QueryTableResponse): if
```

```
len (msg.tables)
```

```
0 : ctx . logger . info ( "There is a free table, attempting to book one now" )
```

table_number

```
msg . tables [ 0 ]
```

request

```
BookTableRequest ( table_number = table_number, time_start = table_query.time_start, duration = table_query.duration, )
```

```
await ctx . send (sender, request)
```

```
else :
```

```
ctx . logger . info ( "No free tables - nothing more to do" ) ctx . storage . set ( "completed" , True )
```

```
@user . on_message (BookTableResponse, replies = set ()) async
```

```
def
```

```
handle_book_response ( ctx : Context ,
```

```
_sender :
```

```
str ,
```

```
msg : BookTableResponse): if msg . success : ctx . logger . info ( "Table reservation was successful" )
```

```
else : ctx . logger . info ( "Table reservation was UNSUCCESSFUL" )
```

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

```
if
```

```
name
```

```
==
```

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

Run the scripts

Run the restaurant agent and then the user agent from different terminals:

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

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

- Restaurant
- [restaurant]: Query: guests=3 time_start=19 duration=2. Available tables: [2].
- User
- [user]: There is a free table, attempting to book one now
- [user]: Table reservation was successful

Was this page helpful?

[Communicating with other agents](#) [How to use agents to verify messages](#)
