Getting started with Fetch.ai and Swarm

Fetch.ai creates a dynamic communication layer that allows you to abstract away components into individualAgents . Agents are microservices programmed to communicate with other agents, and or humans. Written in python, agents are designed to run independently across many and any device.

By usingAgents to integrate withSwarm , you add a layer which can control a narrative and introduce Swarm Agents to a wider ecosystem.

Let's take a look at a simple Swarm example, then see how we can extend this with the uAgents Framework and theBureau class.

A simple Swarm example

Installation

pip install git+ssh://git@github.com/openai/swarm.git

Code

From the Swarm docs, we are going to extend their example:

from swarm import Swarm , Agent

client

Swarm ()

def

transfer_to_agent_b (): return agent_b

agent_a

Agent (name = "Agent A" , instructions = "You are a helpful agent." , functions = [transfer_to_agent_b],)

agent_b

Agent (name = "Agent B" , instructions = "Only speak in Haikus." ,)

response

client . run (agent = agent_a, messages = [{ "role" : "user" , "content" : "I want to talk to agent B." }],)

print (response.messages[- 1]["content"]) There's a few really nice concepts here,functions andinstructions .Functions allow the Agent to call another function or act on the data being sent inclient.run() , this function can also return an agent which passes the message chat to that new agent.Instructions are the prompts; we'll use Agents to update these in the future.

A simple communication with agents

In many other places in our documentation, we have Agent examples for creation, communication and beginner guides. If you've never built using the uAgents Framework before, please have a look at these guides to get yourself started:

- Agents - uAgents Framework
- Creating your first agent
- nstalling the uAgents Framework
- Agents 101

You can read more about Agents and Agent Functions communication in ourguides(opens in a new tab)

The integration with Swarm below is quite simple, so we can skip over a lot of Agents information, for brevity.

Swarm x uAgents Framework

We useBureau for this example, our multi-agent runner that works very similarly to Swarm loop examples. For us,Bureau is a great way of running many Agents on one single device. Our plan here is fairly simple, we want Agents to control the narrative of the Swarm by creating the questions best suited to the response.

So let's extend it.

Self hosted swarm_bureau.py from swarm import Swarm , Agent from swarm import Agent as SwarmAgent from uagents import Agent , Model , Bureau , Context

client

Swarm ()

def

transfer_to_agent_b (): return swarm_agent_b

def

transfer_to_agent_c (): return swarm_agent_c

def

helpful (context_variables): resp =

"You are a quizzical agent. Answer their question in a riddle. Do not answer in a Haiku." ctx = context_variables ["personality"]

if context_variables ["personality"]

else

"" print (ctx) if

"friendly"

in ctx : return

f " { resp } be sure to give them lots of clues, make the riddle not too difficult to answer" else : return

f " { resp } "

swarm_agent_a

SwarmAgent (name = "Agent A" , instructions = "You are a helpful agent." , functions = [transfer_to_agent_b],)

swarm_agent_b

SwarmAgent (name = "Agent B" , instructions = "Only speak in Haikus. Find out what they want" , functions = [transfer_to_agent_c])

swarm_agent_c

SwarmAgent (name = "Agent C" , instructions = helpful,)

swarm_agent_d

SwarmAgent (name = "Question generator" , instructions = "Create a random question to ask someone about any animal")

class

Request (Model): text :

```
str

class

Response ( Model ): text :

str

class

QuestionCreation ( Model ): text :

str

class

QuestionCreated ( Model ): text :

str
```

swarm_uagent

```
Agent (name = 'Swarm' ) trigger_uagent =

Agent (name = 'Trigger' ) question_uagent =

Agent (name = 'Question' )

@swarm_uagent . on_message (Request) async

def

handle_request ( ctx : Context ,

sender :

str ,

request : Request): response = client . run ( agent = swarm_agent_a, messages = [{ "role" : "user" , "content" :
request.text}], )

await ctx . send (sender, Response (text = response.messages[ - 1 ][ "content" ]))

@trigger_uagent . on_event ( 'startup' ) async

def

trigger_request ( ctx : Context): await ctx . send (swarm_uagent.address, Request (text = "I want to talk to agent B." ))

@trigger_uagent . on_message (Response) async

def

handle_response ( ctx : Context ,

sender :

str ,

response : Response): print ( f "Response from on_message : { response.text } " ) await ctx . send
(question_uagent.address, QuestionCreation (text = "" ))

@trigger_uagent . on_message (QuestionCreated) async

def

handle_response ( ctx : Context ,

sender :

str ,
```

response : Response): print (f "Response from on_message :" , response.text)

response

client . run (agent = swarm_agent_c, messages = [{ "role" : "user" , "content" : response.text}], context_variables = { "user" : "Jessica" , "personality" : "friendly, kind" })

print (response.messages[- 1]["content"])

@question_uagent . on_message (QuestionCreation) async

def

create_question (ctx : Context ,

sender :

str ,

question : QuestionCreation): print ("creating question...")

response

client . run (agent = swarm_agent_d, messages = [{ "role" : "user" , "content" : "Create a random question about any animal" }],)

print (response.messages[- 1]["content"])

await ctx . send (sender, QuestionCreated (text = response.messages[- 1]["content"]))

bureau

Bureau () bureau . add (swarm_uagent) bureau . add (trigger_uagent) bureau . add (question_uagent) bureau . run () There's a lot of code there, but so let's cover the Agent specific bits first:

We importeduagents first:

Self hosted swarm_bureau.py from uagents import Agent , Model , Bureau , Context We defined our message structures:

Self hosted swarm_bureau.py class

Request (Model): text :

str

class

Response (Model): text :

str

class

QuestionCreation (Model): text :

str

class

QuestionCreated (Model): text :

str Within the uAgents Framework this is pretty important, and can be much more complex than the above. We treatMessage classes as our rules of communication, generally in a Request/Response format. It also enforces type that helps keep Agents resilient. For additional information on how Agents do communicate, have a look at the followingresource .

Next, we instantiate our Agents and createmessage handling functions :

Self hosted swarm_bureau.py swarm_uagent =

Agent (name = 'Swarm') trigger_uagent =

Agent (name = 'Trigger') question_uagent =

Agent (name = 'Question')

@swarm_uagent . on_message (Request) async

def

handle_request (ctx : Context ,

sender :

str ,

request : Request): response = client . run (agent = swarm_agent_a, messages = [{ "role" : "user" , "content" : request.text}],)

await ctx . send (sender, Response (text = response.messages[- 1]["content"]))

@trigger_uagent . on_event ('startup') async

def

trigger_request (ctx : Context): await ctx . send (swarm_uagent.address, Request (text = "I want to talk to agent B."))

@trigger_uagent . on_message (Response) async

def

handle_response (ctx : Context ,

sender :

str ,

response : Response): print (f "Response from on_message : { response.text } ") await ctx . send (question_uagent.address, QuestionCreation (text = ""))

@trigger_uagent . on_message (QuestionCreated) async

def

handle_response (ctx : Context ,

sender :

str ,

response : Response): print (f "Response from on_message :" , response.text)

response

client . run (agent = swarm_agent_c, messages = [{ "role" : "user" , "content" : response.text}], context_variables = { "user" : "Jessica" , "personality" : "friendly, kind" })

print (response.messages[- 1]["content"])

@question_uagent . on_message (QuestionCreation) async

def

create_question (ctx : Context ,

sender :

str ,

question : QuestionCreation): print ("creating question...")

response

client . run (agent = swarm_agent_d, messages = [{ "role" : "user" , "content" : "Create a random question about any animal" }],)

print (response.messages[- 1]["content"])

await ctx . send (sender, QuestionCreated (text = response.messages[- 1]["content"]))

bureau

Bureau () bureau . add (swarm_uagent) bureau . add (trigger_uagent) bureau . add (question_uagent) bureau . run () The important thing to note is that Agents have specific functions to handle different message objects being received; the uAgents library in the background calls the correct function based on decorator and args. When an Agent callsawait ctx.send(sender, Message()) that effectively calls another Agent's function to act on the sent data.

We have extended the Swarm example slightly; we want to use context variables but also use an Agent to generate the question on our behalf.

Self hosted swarm_bureau.py def

helpful (context_variables): resp =

"You are a quizzical agent. Answer their question in a riddle. Do not answer in a Haiku." ctx = context_variables ["personality"]

if context_variables ["personality"]

else

"" print (ctx) if

"friendly"

in ctx : return

f " { resp } be sure to give them lots of clues, make the riddle not too difficult to answer" else : return

f " { resp } "

Self hosted swarm_bureau.py swarm_agent_c =

SwarmAgent (name = "Agent C" , instructions = helpful,)

swarm_agent_d

SwarmAgent (name = "Question generator" , instructions = "Create a random question to ask someone about any animal") Thecontext_variable was set in an Agent message handler:

Self hosted swarm_bureau.py @trigger_uagent . on_message (QuestionCreated) async

def

handle_response (ctx : Context ,

sender :

str ,

response : Response): print (f "Response from on_message :" , response.text)

response

client . run (agent = swarm_agent_c, messages = [{ "role" : "user" , "content" : response.text}], context_variables = { "user" : "Jessica" , "personality" : "friendly, kind" })

print (response.messages[- 1]["content"]) Let's now recap the flow:

Expected Output

Runpoetry run python langchain_agent_two.py first and thenpoetry run python langchain_agent_one.py .

You should get something similar to the following for the bureau:

INFO:httpx:HTTP Request: POST https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK" Response from on_message : To have a chat with Agent B, Look to the hive with golden key. If none can find, then change the tone, And talk to shadows made of stone. creating question... INFO:httpx:HTTP Request: POST https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK" If you could transform into any animal for a day, which one would you choose and why? INFO: [bureau]: Starting server on http://0.0.0.0:8000 (Press CTRL+C to quit) Response from on_message : If you could transform into any animal for a day, which one would you choose and why? friendly, kind INFO:httpx:HTTP Request: POST https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK" In the sky so high, I soar with might, With wings so vast, I dance with light. I see the world in colors bright, A fish below, a mouse in flight. Majestic, noble, called king of skies, Which creature would I be in disguise?

Next steps

This has been a brief introduction intoSwarm andFetch.ai .

For further reading on how and where we use other OpenAI, technology take a look at the following resource forRAG Agents(opens in a new tab) .

Last updated on October 22, 2024

Was this page helpful?

You can also leave detailed feedbackon Github

Multi-agent workflows with Fetch.ai x Langchain Examples

On This Page

- A simple Swarm example
- Installation
- Code
- A simple communication with agents
- Swarm x uAgents Framework
- Expected Output
- Next steps
- Edit this page on github(opens in a new tab)