Getting started with Fetch.ai x Langchain

Fetch.ai creates a dynamic communication layer that allows you to abstract away components into individualAgents . Agents are micro-services that are programmed to communicate with other agents, and or humans. By usingAgents to represent different parts of yourLangchain program you give your project the option to be used byother parties for economic benefit.

Let's take a look at a simple Langchain example, then see how we can extend this with agents.

A simple langchain example

Let's create a simple script that can find any information in a PDF. Using a document loader from Langchain, and FAISS vector store along with OpenAI, we can load the PDF, useFAISS to create a vector store,open_ai to create embeddings on the documents, and then useFAISS to do a similarity search. Quite complicated for a small example, but it is only a handful of lines of code:

from langchain_community . document_loaders import PyPDFLoader import os from langchain_community . vectorstores import FAISS from langchain_openai import OpenAIEmbeddings

openai_api_key

os . environ ['OPENAI_API_KEY']

loader

PyPDFLoader ("./your-pdf.pdf") pages = loader . load_and_split ()

faiss_index

FAISS . from_documents (pages, OpenAIEmbeddings (openai_api_key = openai_api_key))

docs

faiss_index . similarity_search ("what problem does fetch solve?" , k = 2) for doc in docs : print (str (doc.metadata["page"]) +

":" , doc.page_content[: 600]) However, there is a lot of smaller bits happening there. If we use agents for each step, then other agents can use those pieces of code .

A simple communication with agents

Fetch.ai has the concept of an agent which at a base level an agent cannot do what Langchain does, however an agent is the component that links them together.

You can read more about agents communication in ourguides

Let's install what we need:

poetry

init poetry

add

uagents Check out more detailed instructions forinstallation ofuagents library on your end.

First Agent

Our first agent is simple; it sends a message every two seconds to a static address. When this agent receives a message, it prints that to log:

Self hosted first_agent.py from uagents import Agent , Context , Model from uagents . setup import fund_agent_if_low

class

Message (Model): message :

str

RECIPIENT_ADDRESS

"agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp"

agent

Agent (name = "agent" , port = 8000 , seed = "" , endpoint = ["http://127.0.0.1:8000/submit"],)

fund_agent_if_low (agent.wallet. address ())

@agent . on_interval (period = 2.0) async

def

send_message (ctx : Context): await ctx . send (RECIPIENT_ADDRESS, Message (message = "hello there"))

@agent . on_message (model = Message) async

def

message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

if

name

==

"**main**" : agent . run () This first agent introduces a few core concepts you will need to be aware of when creating any agent.

Agents are defined with theAgent class:

Self hosted first_agent.py agent =

Agent (name = "agent" , port = 8000 , seed = "" , endpoint = ["http://127.0.0.1:8000/submit"],) Aseed is a unique phrase whichuagents library uses to create a unique private key pair for your agent. If you change yourseed you may lose access to previous messages, and also, the agent's address registered to the Almanac will change subsequently. Theport allows us to define a local port for messages to be received. Theendpoint defines the path to the in-built Rest API. Thename defines the name of the agent.

There are more options for theAgent class; seeAgent Class for further reference.

We then need to define our communication model:

Self hosted first_agent.py class

Message (Model): message :

str TheModel defines the object sent from agent to agent and represents the type of messages the agent is able to handle. For explicit communication, both agents must handle the sameModel class.Model is the base class that inherits from Pydantic BaseModel.

With thefund_agent_if_low(agent.wallet.address()) function, agents will ultimately pay for discoverability as the economy of agents matures. There is a placeholder for registration here.

Finally, agents have two decorated functions.

The first one is theagent.on_interval() function. This one sends a message every 2 seconds.ctx.send() has the args ofdestination_address andMessage which we defined earlier.

Self hosted first_agent.py @agent . on_interval (period = 2.0) async

def

send_message (ctx : Context): await ctx . send (RECIPIENT_ADDRESS, Message (message = "hello there")) The second one isagent.on_message() which is a little different; when the agent receives a message at theendpoint we defined earlier, theuagent library unpacks the message and triggers any function which handles that message; in our case, theagent.on_message() function:

Self hosted first_agent.py @agent . on_message (model = Message) async

def

message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

Second Agent

Agent two doesn't do anything different to agent one; it has different args for the Agent instantiation, and instead of sending a messageon_event("startup") , agent two just logs its address to screen. Whenever agent two receives a message matchingMessage data model, it will send a response to the sender.

Self hosted second_agent.py from uagents . setup import fund_agent_if_low from uagents import Agent , Context , Model

class

Message (Model): message :

str

agent

Agent (name = "agent 2" , port = 8001 , seed = "" , endpoint = ["http://127.0.0.1:8001/submit"],)

fund_agent_if_low (agent.wallet. address ())

@agent . on_event ("startup") async

def

start (ctx : Context): ctx . logger . info (f "agent address is { agent.address } ")

@agent . on_message (model = Message) async

def

message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

await ctx . send (sender, Message (message = "hello there"))

if

name

==

"**main**" : agent . run () Okay, let's now run these agents.

Running the agents

Let's run the second agent's script first using this command:poetry run python second_agent.py

We must run the second agent first to get its unique address . This is shown in output in the log. Let's updatefirst_agent.py script by filling theRECIPIENT_ADDRESS field with the address of the second agent from of the output we previously got by runningsecond_agent.py script.

Updatedfirst_agent.py script sample:

first_agent.py from uagents import Agent , Context , Model from uagents . setup import fund_agent_if_low

class

Message (Model): message :

bool

RECIPIENT_ADDRESS

"agent...."

agent

Agent (... Then, let's run the script for the first agent using this command:poetry run python first_agent.py

Great! You should now be seeing some log out output with our messages being displayed.

Output

- First Agent
- :
- INFO: [agent]: Registering on almanac contract...
- INFO: [agent]: Registering on almanac contract...complete
- INFO: [agent]: Starting server on http://0.0.0.0:8000 (Press CTRL+C to quit)
- INFO: [agent]: Received message from agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp: hello there
- INFO: [agent]: Received message from agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp: hello there
- INFO: [agent]: Received message from agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp: hello there
- Second Agent
- :
- INFO: [agent 2]: Registering on almanac contract...
- INFO: [agent 2]: Registering on almanac contract...complete
- INFO: [agent 2]: agent address is agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp
- INFO: [agent 2]: Starting server on http://0.0.0.0:8001 (Press CTRL+C to quit)

Wrapping them together - Building a service

Let's go further now and change our agents scripts by splitting the logic of the Langchain example above. Let's have one agent that sends a PDF path and questions it wants answered about that PDF by the other agent. Conversely, the other agent returns information on the PDF based on the questions asked by using Langchain tools.

Agent one: providing PDF and requesting information

This agent sendsDocumentUnderstanding model which contains a local path to a PDF, and a question that the other agent must answer about the PDF. It's a small update on our first agent script.

However now,.on_message(model=DocumentsResponse) expects aDocumentsResponse object instead of a string.

To learn more about communication with other agents check out the followingGuide

Self hosted langchain_agent_one.py from uagents import Agent , Context , Protocol , Model from ai_engine import UAgentResponse , UAgentResponseType from typing import List

class

DocumentUnderstanding (Model): pdf_path :

str question :

str

class

DocumentsResponse (Model): learnings : List

agent

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

print ("uAgent address: " , agent.address) summary_protocol =

Protocol ("Text Summarizer")

RECIPIENT_PDF_AGENT

""

@agent . on_event ("startup") async

def

on_startup (ctx : Context): await ctx . send (RECIPIENT_PDF_AGENT, DocumentUnderstanding (pdf_path = "../a-little-story.pdf" , question = "What's the synopsis?"))

@agent . on_message (model = DocumentsResponse) async

def

document_load (ctx : Context ,

sender :

str ,

msg : DocumentsResponse): ctx . logger . info (msg.learnings)

agent . include (summary_protocol, publish_manifest = True) agent . run ()

Agent two: wrapping the Langchain bits

Agent two defines the same models as agent one, but this time, it wraps the logic for the Langchain PDF question in thedocument_load() function, which is decorated with.on_message(model=DocumentUnderstanding, replies=DocumentsResponse) . You can specify areplies argument in youron_message decorators; this is useful for being more explicit with communication.

Self hosted langchain_agent_two.py from langchain_community . document_loaders import PyPDFLoader import os from langchain_community . vectorstores import FAISS from langchain_openai import OpenAIEmbeddings from uagents import Agent , Context , Protocol , Model from typing import List

class

DocumentUnderstanding (Model): pdf_path :

str question :

str

class

DocumentsResponse (Model): learnings : List

pdf_questioning_agent

Agent (name = "pdf_questioning_agent" , seed = "" , port = 8003 , endpoint = ["http://127.0.0.1:8003/submit"],)

print ("uAgent address: " , pdf_questioning_agent.address) pdf_loader_protocol =

Protocol ("Text Summariser")

@pdf_questioning_agent . on_message (model = DocumentUnderstanding, replies = DocumentsResponse) async

def

document_load (ctx : Context ,

sender :

str ,

msg : DocumentUnderstanding): loader =

PyPDFLoader (msg.pdf_path) pages = loader . load_and_split () openai_api_key = os . environ ['OPENAI_API_KEY'] learnings = []

faiss_index

FAISS . from_documents (pages, OpenAIEmbeddings (openai_api_key = openai_api_key))

docs

faiss_index . similarity_search (msg.question, k = 2)

for doc in docs : learnings . append (str (doc.metadata["page"]) +

":"

+ doc.page_content[: 600])

await ctx . send (sender, DocumentsResponse (learnings = learnings))

pdf_questioning_agent . include (pdf_loader_protocol, publish_manifest = True) pdf_questioning_agent . run () With these agents now being defined, it is time to run them. Let's run Agent two first to get its address and then update Agent one to send a message to it by filling theRECIPIENT_PDF_AGENT field in-line.

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 each agent:

- Langchain Agent 1
- :
- uAgent address agent: agent1qv9qmj3ug83vcrg774g2quz0urmlyqlmzh6a5t3r88q3neejlrffz405p7x
- INFO: [find_in_pdf]: Manifest published successfully: Text Summarizer
- INFO: [find_in_pdf]: Registration on Almanac API successful
- INFO: [find_in_pdf]: Almanac contract registration is up to date!
- INFO: [find_in_pdf]: Starting server on http://0.0.0.0:8001 (Press CTRL+C to quit)
- INFO: [find_in_pdf]: ['0: This is a simple story about two ...]
- Langchain Agent 2
- :
- uAgent address: agent1qfwfpz6dpyzvz0f0tgxax58fpppaknnqm99fpggmm2wffjcxgqe8sn4cwx3
- INFO: [pdf_questioning_agent]: Manifest published successfully: Text Summarizer
- INFO: [pdf_questioning_agent]: Registration on Almanac API successful
- INFO: [pdf_questioning_agent]: Almanac contract registration is up to date!
- INFO: [pdf_questioning_agent]: Starting server on http://0.0.0.0:8003 (Press CTRL+C to quit)
- INFO:httpx:HTTP Request: POST https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"
- INFO:faiss.loader:Loading faiss with AVX2 support.
- INFO:faiss.loader:Successfully loaded faiss with AVX2 support.
- INFO:httpx:HTTP Request: POST https://api.openai.com/v1/embeddings "HTTP/1.1 200 OK"

Next steps

In thenext part of this introduction, we will create a multi-agent workflow where we split the logic of the PDF agent into two

more agents: the first one which verifies a PDF, loads and then splits the PDF and the second one which uses FAISS to do the similarity search.

Last updated on October 17, 2024

Was this page helpful?

You can also leave detailed feedback on Github

Creating an Agent with Crewai | Multi-agent workflows with Fetch.ai x Langchain

On This Page