AI Agents 101

Overview

Welcome toAI Agents 101 ! This course is designed to introduce you to the development of AI agents, providing a comprehensive guide from foundational concepts to practical implementation. Whether you're a beginner in programming or an experienced developer, this course caters to various skill levels, offering a pathway to create increasingly sophisticated AI agents and explore diverse use cases.

If you encounter uncertainties or have questions about specific terms or topics throughout the course, our support team is available onDiscord ↗(opens in a new tab) to assist you.

Introduction to AI Agents

In this course, you'll delve into the world of AI Agents using theAgents Framework ↗ . AI Agents are programs able to operate autonomously within decentralized landscapes, aligned with user-defined objectives. These agents have the ability to connect, search, transact, establish dynamic markets and so on. By leveraging artificial intelligence, API calls, blockchain technology, and business logic, AI Agents automate multiple workflows. The aim is to facilitate interactions with their environment and other networked agents without human intervention.

Set up your development environment

Prerequisites

Before embarking on this course, ensure your machine meets the following requirements:

1. Python 3.8+
2. : download and install Python fromPython's official website ↗(opens in a new tab)
3. Preferred IDE
4. : Visual Studio Code or PyCharm (alternative options like Notepad are feasible).

Set up development tools

Installing Homebrew

Homebrew streamlines software installations on MacOS via the command line. To install and update Homebrew, execute the following commands:

/bin/bash -c "(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)" You can verify ithere ↗(opens in a new tab) . Let's then ensure Homebrew is updated:

brew update ℹ For more information on Homebrew explore theirwebsite ↗(opens in a new tab) .

Installing PyEnv

Now, you need to installPyEnv . It is a simple tool to manage multiple versions of Python. Run:

brew install pyenv Once you have installed PyEnv you can configure the shell environment:

echo 'export PYENV_ROOT="HOME/.pyenv"' >> ~/.zshrc echo 'command -v pyenv >/dev/null || export PATH="PYENV_ROOT/bin:PATH"' >> ~/.zshrc echo 'eval "(pyenv init -)"' >> ~/.zshrc ℹ These commands configure your shell environment (specifically theZsh shell) to work with PyEnv. These commands set up environment variables, modify the PATH, and initialize PyEnv so that you can easily manage and switch between different Python versions. You can verify all stepshere ↗(opens in a new tab) . You are now ready toinstall Python if you haven't done it yet. You need to install a version of Python 3.8 or above (for this example, we use version 3.10):

pyenv install 3.10 You can get help or check a command insights by running:

pyenv help Let's now ensure theglobal version of Python you are working with is not the default one in the system. Run:

pyenv global 3.10 # this sets the global interpreter pyenv versions # this verifies if it is set up correctly

Installing Poetry

You now need to installPoetry . Poetry is used for managing Python project dependencies, handling virtual environments, packaging, and publishing Python libraries or applications.

You can install Poetry by running the following command:

curl -sSL https://install.python-poetry.org | python3 - ℹ If you would like to learn more about Poetry, visit thewebsite ↗(opens in a new tab) for further information.

Initialize your project with Poetry

You now have all necessary tools installed. You are ready to initialize your project! Let's create a working directory and initialize Poetry .

First, you need to create a working directory for your project usingmkdir command. Then, you will need to change directory to this one, usingcd command:

mkdir development/agent-demo cd development/agent-demo You can ensure you are in the correct directory by checking your current path:

pwd

Example output: /Users/Jessica/Documents

If you are happy with the install location for the project, go ahead andinitialize Poetry :

poetry init ℹ Follow the setup wizard to provide details including project name, version, author, license, and select dependencies (e.g.,uagents). Once you complete the initialization, run:

poetry install This command will install the dependencies specified in thepyproject.toml file.

Congratulations! You've completed the installation process. You're now all set to embark on creating your first AI Agent!

Overview of the uAgents Framework

TheuAgents Framework ↗ provides the essential infrastructure for creating and deploying AI Agents within Fetch.ai Ecosystem. It offers a comprehensive toolkit, protocols, and functionalities crucial for developing autonomous agents. The uAgents Framework is integrated with several components of the Fetch.ai Ecosystem, including theAI Engine ↗ , theAlmanac contract ↗ and theAgentverse ↗ .

ℹ Check out the officialuAgents Python package ↗(opens in a new tab)and start developing your ideas using this library!

Core concepts

Addresses

AI Agents are identified bytwo types of addresses , serving as identifiers for each agent within the Fetch.ai Ecosystem:

1. uAgent Address
2. : the primary agent identifier; it allows it to interact with other agents, exchange messages, and participate in decentralized network activities
3. Fetch Address
4. : this cryptographic public address is linked to the agent and its wallet on the Fetch.ai blockchain; it enables various functionalities, including interacting with theFetch ledger ↗
5. , registering in the Almanac contract and performing operations, including token or asset transfers on the blockchain.

ℹ Check out theGetting an agent addresses guide to learn more on this topic.

Storage

AI Agents can store information inJSON files that they can freely retrieve when needed. Storage plays a critical role in maintaining agent state, retaining past interactions, and facilitating informed decision-making based on historical data.

ℹ Discover more about agents' storage functions through ourUsing agents storage function ↗ guide or theagents storage ↗ documentation. If you are new to JSON, please see an examplehere ↗(opens in a new tab).

Protocols

The uAgents Framework provides comprehensive support for the organization of message types and their handlers withinprotocols . Protocols are sets of rules that govern the transmission, reception and interpretation of data between agents. They define the communication format, timing, sequence and error handling. Protocols enable standardized

communication and ensure accurate and reliable data exchange. Agents using the same protocol can communicate directly with each other.

ℹ If you want to become an expert in Fetch.ai's agent technology, we strongly recommend that you check out protocols ↗ documentation.

Exchange protocol

TheExchange Protocol facilitates efficient communication between agents by using standardized messaging techniques. Messages are packed in envelopes which are encoded and transmitted to specific endpoints via HTTP. Messages consist of key-value pairs (in JSON format) and are packed in envelopes with metadata.Envelopes contain:

- Sender and recipient addresses.
- Message schema.
- Payload.
- Expiration time.
- Signature for authentication.

The exchange protocol uses a standardizedHTTP 1.1 POST /submit endpoint for message processing and expects JSON formatted data. These details ensure consistent and standardized communication within the Fetch.ai agents ecosystem.

ℹ For more information, see ourExchange Protocol ↗ documentation.

Almanac contract, registering, searching and discovery

TheAlmanac ↗ contract is an important component in Fetch.ai Ecosystem. It operates as a blockchain-based repository where agents register, exchange information and establish communication. Registration in the Almanac is mandatory so that agents can participate in remote interactions and become discoverable through theAgentverse Explorer ↗ .

Agents keep their registrations up-to-date within certain block limits to ensure the accuracy and relevance of their data. Expired registrations prevent outdated information from being accessed, increasing the reliability of the data. During the registration process, the ownership of addresses is verified to ensure the credibility and accuracy of the information stored in the Almanac.

The agents registered in the Almanac provide service endpoints for remote communication, which contain weighted parameters that enable effective interaction. As a central hub, the Almanac facilitates the discovery of endpoints based on these assigned weights. This structured approach promotes efficient agent interactions and a robust environment for the development of AI agents within Fetch.ai's decentralized network.

ℹ Check out theRegistering in the Almanac contract ↗ guide and theRegistration and endpoints weighting ↗ for additional information on the registration process.

Coding and implementation

Create your first agent

Creating your first agent is a straightforward process. First of all, we need to create a Python file for this example. We can do this by running:touch alice_agent.py

We can now code our agent. Let's start by importing the required modules. In our case, we would need to import theAgent module from theuagents library, and proceed to instantiate the agent by providing aname . The following code exemplifies the creation of the simplest possible agent:

Import the required classes

from uagents import Agent

Initialize your first Agent and give it a name

alice

Agent (name = "alice" , seed = "alice recovery phrase") You can run this withpoetry run python alice_agent.py but it won't do much, yet!

ℹ If you're not too familiar with classes in Python, take a look w3schools Python classes ↗(opens in a new tab).

Create a second agent and start an interaction

Let's get Alice to do something! We're going to get Alice, on start up, to introduce itself and provide its address by printing both, on the terminal window. We can add aon_event("startup") decorator to make the agent run the defined function when it is run.

Let's updatealice_agent.py with the code snippet below:

Import the required classes

from uagents import Agent , Context

agent

Agent (name = "alice")

Provide your Agent with a job

@agent . on_event ("startup") async

def

introduce_agent (ctx : Context): ctx . logger . info (f "Hello, I'm agent { ctx.name } and my address is { ctx.address } .")

This constructor simply ensure that only this script is running

if

name

==

"**main**" : agent . run () Decorators are moderately advanced in Python, but in this guide all you need to know is that they're there so that the agent knows how to act on the declared decorated functions.

ℹ To find out more about decorators, take a look atPrimer on Python decorators ↗(opens in a new tab). In our example above, theon_event() decorator specifies that the agent should run anintroduce_agent() function when the agent starts up. This function will then return a message presenting the agent with its name and address by using theContext class, which is used to retrieve the agent's related name and address using thectx.name andctx.address methods.

Let's run thealice_agent.py script again. Run:poetry run python alice_agent.py

This time, the output will be:

Hello, I'm agent alice and my address is agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t.

Printing agent's addresses

Sometimes, we just need to see what a value looks like. We can quickly do so in Python by using theprint() in-built function. As we have mentioned earlier, every AI Agent is identified by two addresses within the uAgents Framework:uAgent andFetch Network addresses. We have showed above how to check for an uAgent address using thectx.address method of theContext class. Let's now print them in the console to see their differences using theprint() function:

from uagents import Agent

alice

Agent (name = "alice")

print ("uAgent address: " , alice.address) print ("Fetch network address: " , alice.wallet. address ()) The output of the above would look similar to:

uAgent address: agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t Fetch network address: fetch1454hu0n9eszzg8p7mvan3ep7484jxl5mkf9phg ℹ Checkout ourGetting an agent addresses guide for an in-depth understanding of these topics.

Agents and interval tasks

Interval tasks are tasks or set of instructions executed at predefined time intervals. These are useful for automating repetitive tasks, scheduling background processes, or managing periodic activities in applications. Setting up interval tasks for agents is a great way to harness their potential and streamline processes including bidding, searching, data processing, job scheduling and more.

If you want to create an interval task, you need to use anon_interval() decorator to set up an interval with a timer that triggers the task repetition. As an introductory example, we can consider an agent periodically printing hello and its name on the console. In this case, we apply the decorator to asay_hello() function which will be repeated at the specified time interval:

from uagents import Agent , Context

agent

Agent (name = "agent" , seed = "alice recovery phrase")

@agent . on_interval (period = 2.0) async

def

say_hello (ctx : Context): ctx . logger . info (f 'hello, my name is { ctx.name } ')

if

name

==

"**main**" : agent . run () The output will be printed on your terminal every 2 seconds using thectx.logger.info() method of theContext class.

ℹ TheContext class in theuagents library plays a central role by overseeing message handling and serving as a central hub for essential agent functionalities such as storage, wallet, ledger, and identity management. The output would look as follows:

hello, my name is agent hello, my name is agent hello, my name is agent

Agent interactions and interval tasks

We can now introduce a second agent to demonstrate how two agents can interact with one another. Considering the previous example, we create two agents, each one with a name and seed phrase, and enable them to periodically engage with each other. We need to introduce theBureau class which enables agents, within the same program, to be run together from the same script.

Let's create a new Python file for these agents:touch duo_agent.py .

The script will look as follows:

from uagents import Agent , Context , Bureau

alice

Agent (name = "alice" , seed = "alice recovery phrase") bob =

Agent (name = "bob" , seed = "bob recovery phrase")

@alice . on_interval (period = 2.0) async

def

say_hello (ctx : Context): ctx . logger . info (f 'Hello, my name is { ctx.name } ')

@bob . on_interval (period = 2.0) async

def

say_hello (ctx : Context): ctx . logger . info (f 'Hello, my name is { ctx.name } ')

bureau

Bureau () bureau . add (alice) bureau . add (bob)

if

name

==

"**main**" : bureau . run () You can now run the script:poetry run python duo_agent.py .

The output would look as follows:

[alice] Hello, my name is alice [bob] Hello, my name is bob [alice] Hello, my name is alice [bob] Hello, my name is bob [alice] Hello, my name is alice [bob] Hello, my name is bob

Agent communication

We can now show how to enable effective communication between different agents. To do so, we need to introduce theModel class which allows to create a structured message format for messages to be exchanged between the agents.

Let's create a new script for this example:touch agent_communication.py .

We can now define theMessage data model and our two agents:

from uagents import Agent , Bureau , Context , Model

class

Message (Model): message :

str

alice

Agent (name = "alice" , seed = "alice recovery phrase") bob =

Agent (name = "bob" , seed = "bob recovery phrase") We need to define a function foralice to send messages tobob periodically. We can do this by defining asend_message() function using theContext class and makealice send a message tobob on an interval:

@alice . on_interval (period = 3.0) async

def

send_message (ctx : Context): await ctx . send (bob.address, Message (message = "hello there bob")) We then need a way forbob to receive these messages. We can do this by creating a function forbob to handle all incoming messages from other agents. We can do this with aon_message() decorator that will activate themessage_handler() function oncebob receives a message matching theMessage data model we previously defined:

@bob . on_message (model = Message) async

def

bob_message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ") await ctx . send (alice.address, Message (message = "hello there alice")) We then need to define a message handler function foralice to handle response messages frombob . We do so using aon_message() decorator:

@alice . on_message (model = Message) async

def

alice_message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ") Finally, we need to add both agents to theBureau in order to run them from the same script:

bureau

Bureau () bureau . add (alice) bureau . add (bob)

if

name

==

"**main**" : bureau . run () We can now run the script. Just run:poetry run python agent_communication.py in your terminal.

The output would look as follows:

[alice]: Received message from agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94pqp2e94cylu6haugt2hd7j9vequ7: hello there alice [bob]: Received message from agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t: hello there bob [alice]: Received message from agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94pqp2e94cylu6haugt2hd7j9vequ7: hello there alice [bob]: Received message from agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t: hello there bob [alice]: Received message from agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94pqp2e94cylu6haugt2hd7j9vequ7: hello there alice [bob]: Received message from agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t: hello there bob

Enabling search and discovery for your Agent (Almanac registration)

Agent registration in the Almanac contract is a key feature which enables discoverability of agents as well enabled remote agent communication. To register, agents must pay a small fee. Therefore your agents need to have funds available in their Fetch wallet address. Luckily in this demo we utilise the testnet environment to simulate real world transactions.

ℹ When using the testnet, you can use the functionfund_agent_if_low() to fund your agent. We first import modules, then initialize our agents and include a function ensuring that agents have non-zero balances in their wallets. This function will check if you have enough tokens to register in the Almanac. If not, it will add tokens to your Fetch wallet address. AI Agents can communicate by querying the Almanac and retrieving an HTTP[endpoint ↗](#) from the recipient agent. Therefore, we need to specify the service endpoints when defining an agent at registration.

ℹ HTTP (Hypertext Transfer Protocol) service endpoints are specific locations or URLs (Uniform Resource Locators) on a web server where clients can send HTTP requests to interact with resources or services provided by the server. These endpoints define the entry points for various operations or functions offered by a web service or application. Let's create a Python script for this example:touch almanac_registration.py .

We will have what follows:

almanac_registration.py from uagents . setup import fund_agent_if_low from uagents import Agent , Context , Protocol

alice

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

fund_agent_if_low (alice.wallet. address ())

@alice . on_interval (period = 3) async

def

hi (ctx : Context): ctx . logger . info (f "Hello")

alice . run () Here, we defined a local http address but you could also define a remote address to allow agent communication over different machines through the internet. Importantly, make sure to add a seed phrase to your agent so you don't have to fund different addresses each time you run your agent.

ℹ A seed phrase is a series of random words (typically 12 or 24) that provide the data needed to recover a lost or broken crypto wallet. It is also known as a mnemonic phrase and is best understood as a security measure for self-custody of digital assets. Agents have a crypto wallet address, and having the seed phrase enables the restoration of an agent's wallet address. To run the script use thepoetry run python almanac_registration.py command.

Remote agent communication

AI Agents can also interactremotely . To achieve a remote communication, we simply need an agent's address and query the rest of its information in the Almanac contract. You can create two agents operating on separate ports and terminals within the same device; this mirrors a real-world scenario in which agents communicate across different geographic locations.

In this example, we provide scripts for two agents. To establish a line of remote communication, both agents need to be registered on the Almanac contract and need to have non-zero balances in their Fetch wallet addresses.

We can start withalice agent. Let's create a Python script for it:touch remote_alice.py .

We first import the required modules. We then use theModel class to define aMessage data model for messages to be exchanged between our agents. We also need to provide Bob's address as a recipient address for reference. We can then create our agentalice , by providing the needed information for registration. We need to make sure it has enough balance in its wallet. We then proceed and define its functions. The script would be as follows:

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

class

Message (Model): message :

str

RECIPIENT_ADDRESS

"agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50"

alice

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

fund_agent_if_low (alice.wallet. address ())

@alice . on_interval (period = 2.0) async

def

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

@alice . 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**" : alice . run () Similarly, we need to define a script forbob so to create a remote communication withalice agent. Let's create a Python script for it:touch remote_bob.py . Instead of creating and manually writing out the same script we can copy and rename Alice's script and modify the agent'sname ,seed ,port , decorator as well as the message content:

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

class

Message (Model): message :

str

bob

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

fund_agent_if_low (bob.wallet. address ())

@bob . 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 alice"))

if

name

==

"**main**" : bob . run () In different terminal windows, first runremote_bob.py and thenremote_alice.py . They will register automatically in the Almanac contract using their funds. The received messages will print out in each terminal. In order to run the two agents in parallel terminals use thepoetry run python remote_alice.py andpoetry run python remote_bob.py . The expected output would be:

Alice :

[alice]: Received message from agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50: hello there alice [alice]: Received message from agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50: hello there alice [alice]: Received message from agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50: hello there alice Bob :

[bob]: Received message from agent1qdp9j2ev86k3h5acaayjm8tpx36zv4mjxn05pa2kwesspstzj697xy5vk2a: hello there bob [bob]: Received message from agent1qdp9j2ev86k3h5acaayjm8tpx36zv4mjxn05pa2kwesspstzj697xy5vk2a: hello there bob [bob]: Received message from agent1qdp9j2ev86k3h5acaayjm8tpx36zv4mjxn05pa2kwesspstzj697xy5vk2a: hello there bob

ℹ Checkout ourCommunicating with other agents guide for a deeper explanation of the concepts surrounding AI Agents communication both locally and remotely.

Agents and storage

Agents within the uAgents Framework have the ability to store information locally within a JSON file. This ensures data retrieval as needed. This storage functionality serves as a fundamental component for agents to maintain a state, recollect prior interactions, and base decisions on historical data.

The aim behind integrating storage features is to empower agents to preserve and leverage information over time, facilitating the recollection of past interactions and context for more informed decision-making. This capacity to learn from past experiences enables agents to adapt and refine their behavior and decision processes.

Retrieving or setting storage information within the Framework is achieved through two methods:

- ctx.storage.get()
- for retrieval.
- ctx.storage.set()
- for setting data.

An example is provided below. In this example we have a full script for an agent holding a number, incrementing it by one and saving the new number to its storage.

We first create the Python file containing the script. Run:touch storage.py in your terminal. The script for this example is:

storage.py from uagents import Agent , Context

alice

Agent (name = "alice" , seed = "alice recovery phrase")

@alice . on_interval (period = 1.0) async

def

on_interval (ctx : Context): current_count = ctx . storage . get ("count")

or

0

ctx . logger . info (f "My count is: { current_count } ")

ctx . storage . set ("count" , current_count +

1)

if

name

==

"**main**" : alice . run () We can then run the script:poetry run python storage.py The output would be:

[alice]: My count is: 1 [alice]: My count is: 2 [alice]: My count is: 3 ...

Booking a table at a restaurant

We now want to show how to set up the code to create a **restaurant booking service with twoAI Agents : arestaurant with tables available and auser requesting a table availability.

We can do this by defining 2 specificprotocols : one for table querying (i.e.,Table querying protocol) and one for table booking (i.e.,Table booking protocol). We then need to define two agents,restaurant anduser , which will make use of the protocols to query and book a table.

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

Table querying protocol

Let's start by defining the protocol for querying availability of tables at the restaurant. We start by importing the necessary classes and defining the message data models for types of messages being handled. We then proceed and create an instance of theProtocol class and name itquery_proto :

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 () Here, we defined different messages data models:

- TableStatus
- represents the status of a table and includes the attributes number of seats, start time, and end time.
- QueryTableRequest
- is used for querying table availability. It includes information about the number of guests, start time, and duration of the table request.
- QueryTableResponse
- contains the response to the query table availability. It includes a list of table numbers that are available based on query parameters.
- GetTotalQueries
- is used to request the total number of queries made to the system.
- TotalQueries
- contains the response to the total queries request, including the count of total queries made to the system.

Let's then define the message handlers for thequery_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 ()

pylint: disable=protected-access

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)) Here, thehandle_query_request() function is the message handler function defined using theon_message() decorator. It handles theQueryTableRequest messages and replies with aQueryTableResponse message. The handler processes the table availability query based on the provided parameters, checks the table status 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. On the other hand,handle_get_total_queries() is the message handler function defined using theon_query() decorator. It handles theGetTotalQueries query and replies with aTotalQueries 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.

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 ()

pylint: disable=protected-access

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. We first need to import the necessary classes and define the message data models. In this case, the booking protocol consists of two message models:BookTableRequest andBookTableResponse . Then, create an instance of theProtocol class and name itbook_proto :

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 () * BookTableRequest * represents the request to book a table. It includes attributes:table_number * to be booked,time_start * of the booking, and the duration of the booking. * BookTableResponse * contains the response to the table booking request. It includes a boolean attribute success indicating whether the booking was successful or not.

Let's now define the message handler function:

@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 ()

pylint: disable=protected-access

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)) Thehandle_book_request() handler first retrieves table statuses from the agent's storage and converts them into a dictionary with integer keys (table numbers) andTableStatus values. TheTableStatus class is imported from the query module. Next, the handler gets the table associated with the requestedtable_number from the tables dictionary. The handler checks if the requested time_start falls within the availability period of the table. If the table is available for the requested booking duration, the handler sets success toTrue , updates the table'stime_start to reflect the end of the booking, and saves the updated table information in the agent's storage usingctx.storage.set() . If the table is not available for the requested booking, the handler sets success toFalse . The handler sends aBookTableResponse message back to the sender with the success status of the booking using thectx.send() method.

The overall script should be:

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

Let's now move forward and create our restaurant agent in a separate file. In this step, we'll import the essential classes from theuagents library and reintegrate the two protocols we've previously coded. As we define ourrestaurant agent, it's vital to ensure it possesses sufficient funds in its wallet for the registration process. Remember to use thefund_agent_if_low method for this.

rom uagents import Agent from uagents . setup import fund_agent_if_low

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 ()) Let's build therestaurant agent from above protocols and set the table availability information, by also to storing theTABLES information in therestaurant agent storage:

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 () Therestaurant agent is now online and ready to receive messages.

The overall script would be as follow:

restaurant_agent.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 agent

We can now define the script for ouruser agent querying and booking a table at therestaurant .

Once we've imported the necessary classes from theuagents library and the two protocols we previously defined, we also need therestaurant agent's address so for theuser agent to be able to communicate with it:

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

RESTAURANT_ADDRESS

"agent1qw50wcs4nd723ya9j8mwxglnhs2kzzhh0et0yl34vr75hualsyqvqdzl990"

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 ()) Let's then create the table query to generate theQueryTableRequest using therestaurant address. Then, we need to create anon_interval() function which periodically queries therestaurant , asking for the availability of a table given thetable_query parameters:

table_query

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

@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) We then need to define the message handler function for incomingQueryTableResponse messages from therestaurant agent:

@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) Let's then define a function which will handle messages from therestaurant agent on whether the reservation was successful or not:

@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 () The overall script would be:

user_agent.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

"agent1qw50wcs4nd723ya9j8mwxglnhs2kzzhh0et0yl34vr75hualsyqvqdzl990"

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 () We are ready to run the example.

Run therestaurant agent and then theuser agent from different terminals. The output should be as follows:

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

From novice to navigator: your course conclusion and beyond!

We appreciate your active participation in our introductory course on AI agents! The knowledge you've acquired here forms a robust basis for your future AI Agents development endeavors.

Now, it's time to put your newfound skills to work. We invite you to delve deeper into the world of AI agents by exploring our dedicatedAI Agents ↗ documentation andGitHub ↗(opens in a new tab) repository. Also, do not forget to checkout our full list ofAI Agents guides ↗ diving into the development of AI Agents and concepts explained in this introductory course, but in a more detailed manner.

Join ourDiscord ↗(opens in a new tab) and team up with other developers in order to participate in hackathons, collectively build projects, or simply have fun!

There, you can not only star the project but also access valuable resources that will enhance your agent development journey. We look forward to seeing your contributions and witnessing your continued growth in the realm of AI and agent-based systems.

Was this page helpful?