Controller Draft

Introduction

ACA-Py is a Python-based implementation of the Aries architecture that provides developers with an easy-to-use framework for building controllers. Here are a few examples of ACA-Py controller implementations:

DIDComm Agent: The DIDComm Agent is a basic implementation of an ACA-Py controller that provides a simple interface for sending and receiving DIDComm messages. It includes functionality for creating, verifying, and storing DID documents and keys.

Mediator: The Mediator is an ACA-Py controller that provides routing and mediation services for DIDComm messages. It allows users to create and manage their own routing and mediation networks, and includes functionality for registering and discovering mediator endpoints.

Credential Issuer: The Credential Issuer is an ACA-Py controller that provides functionality for issuing verifiable credentials. It includes functionality for creating credential definitions, issuing credentials, and storing and revoking credentials.

Verifier: The Verifier is an ACA-Py controller that provides functionality for verifying verifiable credentials. It includes functionality for verifying proofs, checking revocation status, and extracting data from credentials.

Wallet: The Wallet is an ACA-Py controller that provides functionality for managing user wallets. It includes functionality for creating and storing DID documents and keys, as well as storing and managing verifiable credentials.

Controller Design Examples

issuer.py

This example demonstrates how to create a new schema and credential definition, and then issue a new credential using those definitions. The `AriesAgentController` class provides a convenient interface for interacting with the ACA-Py API by wrapping the HTTP requests and responses in Python objects.

agent.py

This example sets up a simple DIDComm Agent that listens for incoming messages and sends messages to other DIDs.In this example, the `message_handler` coroutine listens for incoming messages and prints them to the console. The `send_message` coroutine sends a message to a specified DID. These coroutines can be extended and customized to handle more complex message handling and sending scenarios.

mediator.py

In this example, the `message_handler` coroutine listens for incoming DIDExchange messages and automatically handles the connection negotiation process. Once a connection is established, the `mediator_register` method is used to add routing information for the connection. This routing information can be used by other agents to route messages through the mediator.

cred-issuer.py

This example sets up a simple Credential Issuer that creates and issues a credential to another party. In this example, the `issue_credential` coroutine creates a credential offer, creates a credential request in response to the offer, creates the credential itself using specified credential values, and sends the credential to the other party using the `issuer_send_credential` method.

verifier.py

This example sets up a simple Verifier that verifies a credential using a predefined credential schema and definition. In this example, the verify_credential coroutine retrieves a credential from the agent's wallet using the `wallet_get_credential` method, verifies the credential using the `verifier_verify_credential` method, and checks the verification result to determine if the credential was successfully verified.

wallet.py

This example sets up a simple Wallet that creates a new DID and stores it in the wallet, and then retrieves all credentials from the wallet using the `wallet_get_all_credentials` method. In this example, the `create_did` coroutine creates a new DID using the `wallet_create_did` method, stores the DID in the wallet using the `wallet_set_public_did` method, and prints out the DID information. The `get_credentials` coroutine retrieves all credentials from the wallet using the `wallet_get_all_credentials` method and prints them out.

app.js

Wallet implementation we wrote earlier with a React frontend using the `react` and `axios` libraries:

In this example, we define a `Wallet` component that uses the useState hook to manage state for the DID information and credentials. We also define two functions, createOracleDid and getCredentials, that use the axios library to send HTTP requests to the server to create an Oracle DID and get all credentials from the wallet, respectively.

When the component mounts, we call the `createOracleDid` function to create an Oracle DID and store it in the wallet. We also render a button to allow the user to create a new Oracle DID, a button to allow the user to get all credentials from the wallet, and a list of the credentials if there are any.

To use this `Wallet` component in your application, you would need to import it and render it in a parent component. You would also need to set up a server that handles the HTTP requests to the ACA-Py API using the `AriesAgentController` class, and expose an API that the client can use to interact with the wallet.

Testing

There are several types of automatic tests that can be included in an ACA-Py controller, including:

Unit tests: These tests focus on individual units of code (e.g., functions or methods) to ensure that they work as expected. Unit tests can be written using a testing framework like `pytest` or uni`ttest.

Integration tests: These tests ensure that different components of the ACA-Py system work together correctly. For example, an integration test might test that a connection can be established between two agents or that a credential can be issued and verified.

End-to-end tests: These tests simulate real-world usage of the ACA-Py system, from the perspective of a user. End-to-end tests can be written using a tool like Selenium to automate interactions with a web application that uses the ACA-Py API.

Performance tests: These tests measure the performance of the ACA-Py system under different loads and conditions. Performance tests can help identify bottlenecks and optimize the system for better performance.

Here are some examples of tests that could be included in an ACA-Py controller:

Unit tests for individual methods in the controller, such as create_connection or issue_credential. Integration tests that verify that connections can be established between different agents and that credentials can be issued and verified. End-to-end tests that simulate a user interacting with a web application that uses the ACA-Py API, such as creating a connection, requesting a proof, and verifying the proof. Performance tests that measure the response time and throughput of the ACA-Py system under different loads and conditions, such as a large number of simultaneous connections or credentials being issued and verified.

mongdodb-test.py

In this example, we've added two tests that use a MongoDB server to insert and find data. We've also included setup and teardown methods to start and stop the ACA-Py agent and to clean up any test data from the MongoDB server after the tests have finished running.

To run these tests, you can use a testing framework like pytest or unittest. For example, to run the tests using unittest, you can run the following command:

```
python -m unittest test_didcomm_agent.py
```