Agent Draft

Introduction

The Aries framework is designed to be modular, and the specific API calls made between the Aries wallet and other components can depend on the implementation and configuration of those components. However, here is a general overview of the API calls that may be involved in a typical Aries DID and VC flow:

1. The Aries wallet sends a request to an Aries agent, asking it to create a new DID. The Aries agent may then forward this request to a DID method implementation, such as the Oracle Blockchain Platform DID method implementation.
2. The Oracle Blockchain Platform DID method implementation generates a new DID for the user, using the OBP blockchain network.
3. The Oracle Blockchain Platform DID method implementation creates a new DID document for the user, which includes the public key associated with the new DID.
4. The Oracle Blockchain Platform DID method implementation stores the DID document on the OBP blockchain network, using the appropriate smart contract or chaincode.
5. The Oracle Blockchain Platform DID method implementation sends the new DID and associated DID document back to the Aries agent.
6. The Aries agent then sends the new DID and associated DID document to the Aries wallet, which can then use the DID for various purposes, such as creating and sharing VCs.

It's worth noting that the Aries framework is designed to be flexible, and different implementations may use different API calls or workflows depending on their specific needs. This overview provides a general idea of the types of API calls that may be involved in an Aries DID and VC flow, but specific implementations may vary.

Required Design Files

These are some of the files that you will need for implementing an Aries Agent on a virtual machine.

- Aries Agent codebase: The codebase for the Aries Agent, which you can download or clone from the appropriate repository on GitHub, such as the Hyperledger Aries Cloud Agent Python (ACA-Py) repository.
- Virtual Machine: You will need a virtual machine to run the Aries Agent. You can use a virtual machine provider like VirtualBox, VMware, or Microsoft Hyper-V to create a virtual machine.
- Operating System: You will need an operating system to run on the virtual machine. ACA-Py is compatible with several operating systems, including Ubuntu, CentOS, and Windows.
- Python: ACA-Py is written in Python, so you will need to install Python on the virtual machine.
- Dependencies: ACA-Py has several dependencies, including libraries for cryptography and networking. You will need to install these dependencies on the virtual machine.
- Configuration file: You will need to create a configuration file for the Aries Agent, which specifies the agent's settings and parameters. This file typically includes information such as the agent's endpoint URL, the agent's public and private keys, and the agent's wallet settings.

- Credential Schemas and Definitions: Depending on the specific use case, you may need to define the schemas and definitions for the credentials that the agent will create and handle.
- Credentials and Verification Policies: You may also need to create and manage the policies that will be used to verify and authenticate the credentials issued by the agent.

Components

The Aries Agent is typically implemented as a set of components, including a message handler, a wallet, and a transport layer. Here is a brief description of each component:

- Message Handler: The message handler is responsible for processing messages that the agent receives from other agents or verifiers. It can also send messages to other agents or verifiers. The message handler is typically implemented using a message queue or a message broker.
- Wallet: The wallet is used to store and manage the agent's cryptographic keys, credentials, and other information. The wallet is typically implemented using a database or a file system.
- Transport Layer: The transport layer is responsible for sending and receiving messages between agents. It can use various transport protocols, such as HTTP, WebSocket, or SMTP.

In addition to these components, the Aries Agent may also include functionality for managing connections, issuing and verifying credentials, and establishing and managing pairwise relationships with other agents.

Agent Design Examples

message-handler.py

This code assumes the use of the Aries Basic Controller, a Python library that provides a simplified interface for interacting with an Aries Agent. The `MessageHandler` class takes an `AriesAgentController` object as a parameter and uses it to listen for incoming messages and send outgoing messages.

In this example, the `listen_for_messages()` method is a coroutine that listens for incoming messages using the `listen_for_message()` method of the `AriesAgentController`. When a new message arrives, the `process_message()` method is called to handle the message. The `process_message()` method extracts the message type and content from the message and uses a series of conditional statements to handle each type of message differently.

In this example, the `MessageHandler` can handle two message types: `did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/ping/1.0/ping` and `did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/basicmessage/1.0/message`. The `ping` message is used to test the connectivity between agents, while the `basicmessage` message is used to exchange simple text messages between agents. For each message type, the code performs the appropriate actions, such as sending a response to a `ping` message or ignoring other message types.

transport-layer.py

This implementation uses the aiohttp library to send and receive HTTP requests to and from the agent's message handler. The `__aenter__` and `__aexit__` methods allow the AriesTransport object to be used as an asynchronous context manager.

The `send_message` method sends a JSON-encoded message to the agent's message handler at the specified endpoint and returns the response as a JSON object. The `receive_message` method waits for a message to be sent to the agent's message handler and returns the message as a JSON object.

config.haml

In this example configuration file, the `default` section specifies the default configuration values for the agent. The `agent_name` setting specifies the name of the agent instance, while the endpoint setting specifies the `endpoint` for the agent's web server. The `public_did`, `seed`, and `verkey` settings specify the agent's public DID, seed value, and verification key, respectively.

The `message_handler` setting specifies the endpoint for the agent's message handler, which is used to process incoming messages. The `wallet_config` and `wallet_credentials` settings specify the configuration and credentials for the agent's wallet, respectively.

The `mediator_config` setting specifies the configuration for a mediator agent, which can be used to facilitate communication between agents. The `ledger_config` setting specifies the configuration for the distributed ledger used by the agent, such as the name of the pool and the path to the genesis transaction file.

The `plugin_config` setting can be used to specify configuration settings for any plugins used by the agent. Finally, the debug setting specifies whether the agent should run in `debug` mode or not.