LearnThings.Online

Becoming a Hyperledger Aries Developer

This course is from edX, scroll down & check “Read More” for more informations.

About this course

Skip About this course

Data is driving our world today. However, we hear about Data breaches and Identity thefts all the time. Trust on the Internet is broken, and it needs to be fixed. As such, it is imperative that we adopt a new approach to identity management, and ensure data security and user privacy through tamper-proof transactions and infrastructures.

Blockchain-based identity management is revolutionizing this space. The tools, libraries, and reusable components that come with the three open source Hyperledger projects, Aries, Indy and Ursa, provide a foundation for distributed applications built on authentic data using a distributed ledger, purpose-built for decentralized identity.

This course focuses on building applications on top of Hyperledger Aries components—the area where you, as a Self-Sovereign identity (SSI) application developer, can have the most impact. While you need a good understanding of Indy (and other ledger/verifiable credential technologies) and a basic idea of what Ursa (the underlying cryptography) is and does, Aries is where you need to dig deep.

What you'll learn

Skip What you'll learn

- Understand the Hyperledger Aries architecture and its components.
- Discuss the DIDComm protocol for peer-to-peer messages.
- Deploy instances of Aries agents and establish a connection between two or more Aries agents.
- Create from scratch or extend Aries agents to add business logic.
- Understand the possibilities available through the implementation of Aries agents.

Welcome!

Introduction and Learning Objectives

Introduction

Welcome to LFS173x – Becoming a Hyperledger Aries Developer!

The three Hyperledger projects, Indy, Aries and Ursa, provide a foundation for distributed applications built on authentic data using a distributed ledger, purpose-built for decentralized identity. Together, they provide tools, libraries, and reusable components for creating and using independent digital identities rooted on blockchains or other distributed ledgers so that they are interoperable across administrative domains, applications, and any other "silo." While this course will mention Indy and Ursa, its main focus will be Aries and the possibilities it brings for building applications on a solid digital foundation of trust. This focus will be explained further in the course but for now, rest assured: if you want to start developing applications that are identity focused and on the blockchain, this is where you need to be.

What You Will Learn

This course is designed to get students from the basics of Trust over IP (ToIP) and self-sovereign identity (SSI)—what you learned about in the last course (LFS172x – *Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries & Ursa*)—to developing code for issuing (and verifying) credentials with your own Aries agent.

Terminology

We use the terms blockchain, ledger, decentralized ledger technology (DLT) and decentralized ledger (DL) interchangeably in this course. While there are precise meanings of those terms in other contexts, in the context of the material covered in this course the differences are not meaningful.

For more definitions, take a look at our course Glossary section.

Labs and Demos

Throughout this course there will be labs and demos that are a key part of the content and that you are strongly advised to complete. The labs and demos are hosted on GitHub so that we can maintain them easily as the underlying technology evolves. Many will be short interactions between agents. In all cases, you will have access to the underlying code to dig into, run and alter.

For some labs, you won't need anything on your laptop except your browser (and perhaps your mobile phone). For others, you have the option of running the labs in your browser or locally on your own system and we will provide instructions for using a service called *Play with Docker*. Play with Docker allows you to access a terminal command line environment in your browser so you don't have to install everything locally. The downside of using Play with Docker is that you don't have all the code locally to review and update in your local code editor.

When you run the labs locally, you need the following prerequisites installed on your system:

- A terminal command line interface running bash shell.
  – This is built-in for Mac and Linux, and on Windows, the "git-bash" shell comes with git (see installation instructions below).
- Docker, including Docker Compose – Community Edition is fine.
  – If you do not already have Docker installed, open Docker Docs, "Supported Platforms" and then click the link for the installation instructions for your platform.
  – Instructions for installing docker-compose for a variety of platforms can be found here.
- Git
  – This link provides installation instructions for Mac, Linux (including if you are running Linux using VirtualBox) and native Windows (without VirtualBox).

All of the labs that you can run locally use Docker. You can run the labs directly on your own system without Docker, but we don't provide instructions for doing that, and we highly recommend you not try that until you have run through them with Docker. Because of the differences across systems, it's difficult to provide universal instructions, and we've seen many developers spend too much time trying to get everything installed and working right.

The teams we work with only use Docker/containers for development and production deployment. In other cases, developers unfamiliar with (or not interested in) Docker set up their own native development environment. However, doing so is outside the scope of the labs in this course.

Acknowledgements

We have many, many people to thank for this course. The first and foremost is the Linux Foundation for allowing us to share what we know about this evolving and exciting technology. In particular, thank you Flavia and Magda for your guidance and expertise in getting this course online.

We would also like to thank the developers, visionaries and change-seekers in the Hyperledger Indy, Aries and Ursa world—the people we talk to on the weekly calls and daily rocketchats, the people we meet at conferences, and the people who are driving this technology on a daily basis to make the Internet a better place. We would especially like to thank Hyperledger contributors, the Sovrin Foundation, the BC Government VON team and Evernym for their contributions to this ecosystem. As well, a shout out goes to Akiff Manji (@amanji on GitHub) for contributing the Aries ACA-Py Controllers repository and for his updates to the Aries API demo. And to Hannah Curran for her keen editing eye.

While we have tried to be as accurate and timely as possible, these tools and libraries are changing rapidly. There are no doubt mistakes and we own them. Keeping this in mind, we have created a change log on GitHub to track course updates that are needed when mistakes are found in the content and when a major change or shift occurs within the Hyperledger Indy, Aries and Ursa space. If you find an error, a need for a content update, or something in one of the demos doesn't work, please let us know via the GitHub repo.

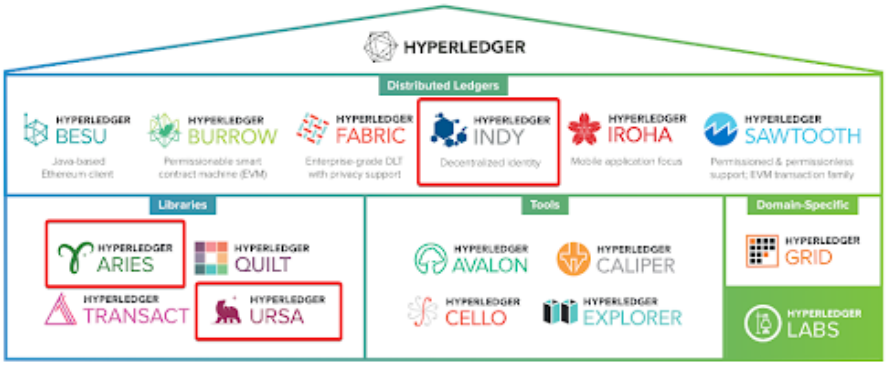
Thank you for taking this course!

Chapter 1. Overview

Chapter Overview

Data breaches. Identity theft. Large companies exploiting our personal information. Trust on the Internet. We read about these Internet issues all the time. Simply put, the Internet is broken and it needs to be fixed.

This is where the Hyperledger Indy, Aries and Ursa projects come in and, we assume, one of the main reasons you are taking this course. The Indy, Aries and Ursa tools, libraries, and reusable components provide a foundation for distributed applications to be built on authentic data using a distributed ledger, purpose-built for decentralized identity (refer to the following image).



The Hyperledger Frameworks and Tools
Licensed under CC BY 4.0

Note: This course is called *Becoming a Hyperledger Aries Developer* because it focuses on building applications on top of Hyperledger Aries components—the area where you, as

an SSI application developer, can have the most impact. Aries builds on Indy and Ursa. While you need to have a good understanding of Indy (and other ledger/verifiable credential technologies) and a basic idea of what Ursa (the underlying cryptography) is and does, Aries is where you need to dig deep.

Learning Objectives

By the end of this chapter you should:

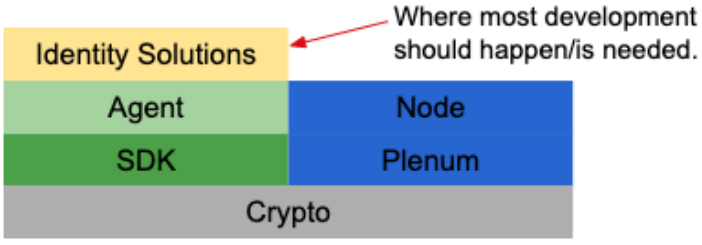
- Understand why this course focuses on Aries (and not Indy or Ursa).
- Understand the problems Aries is trying to fix.
- Know the core concepts behind self-sovereign identity.

Why Focus on Aries Development?

Hyperledger Indy, Aries and Ursa make it possible to:

- Establish a secure, private channel with another person, organization, or IoT thing—like authentication plus a virtual private network—but with no session and no login.
- Send and receive arbitrary messages with high security and privacy.
- Prove things about yourself; receive and validate proofs about other parties.
- Create an agent that proxies/represents you in the cloud or on edge devices.
- Manage your own identity:
  – Authorize or revoke devices.
  – Create, update and revoke keys.

However, you will have relatively little interaction with Indy, and almost none with Ursa, as the vast majority of those working with the Hyperledger identity solutions will build on top of Aries; only those contributing code directly into Indy, Aries and Ursa (e.g. fixing a flaw in a crypto algorithm implementation) will have significant interaction with Indy and Ursa. And here's another big takeaway: while all three projects are focused on decentralized identity, Indy is a specific blockchain, whereas Aries is blockchain-agnostic.



Where Most Development Happens

Why We Need Identity Solutions

In today's world, we are issued **credentials** as documents (for example, our driver's license). When we need to prove who we are, we hand over the document.
The **verifier** looks at the document and attempts to ascertain whether it is valid.
The **holder** of the document cannot choose to only hand over a certain piece of the document but must hand over the entire thing.

A typical paper credential, such as a driver's license, is issued by a government authority (an **issuer**) after you prove to them who you are (usually in person using your passport or birth certificate) and that you are qualified to drive. You then hold this credential (usually in your wallet) and can use it elsewhere whenever you want—for example, when renting a car, in a bank to open up an account or in a bar to prove that you are old enough to drink. When you do that you've **proven** (or **presented**) the credential. That's the **paper credential model**.

Examples of Paper Credentials
By Peter Stokyo
Licensed under CC BY 4.0

The paper credential model (ideally) **proves**:

- Who issued the credential.
- Who holds the credential.
- The claims have not been altered.

The caveat "ideally" is included because of the possibility of forgery in the use of paper credentials. As many university students know, it's pretty easy to get a fake driver's license that can be used when needed to "prove" those same things.

The Verifiable Credential (VC) Model

Enter the VC model, the bread and butter behind decentralized identity, which brings about the possibility of building applications with a solid digital trust foundation.

The verifiable credentials model is the digital version of the paper credentials model. That is:

- An authority decides you are eligible to receive a credential and issues you one.
- You hold your credential in your (digital) wallet.
- At some point, you are asked to prove the claims from the credential.

- You provide a verifiable presentation to the verifier, proving the same things as with a paper credential.

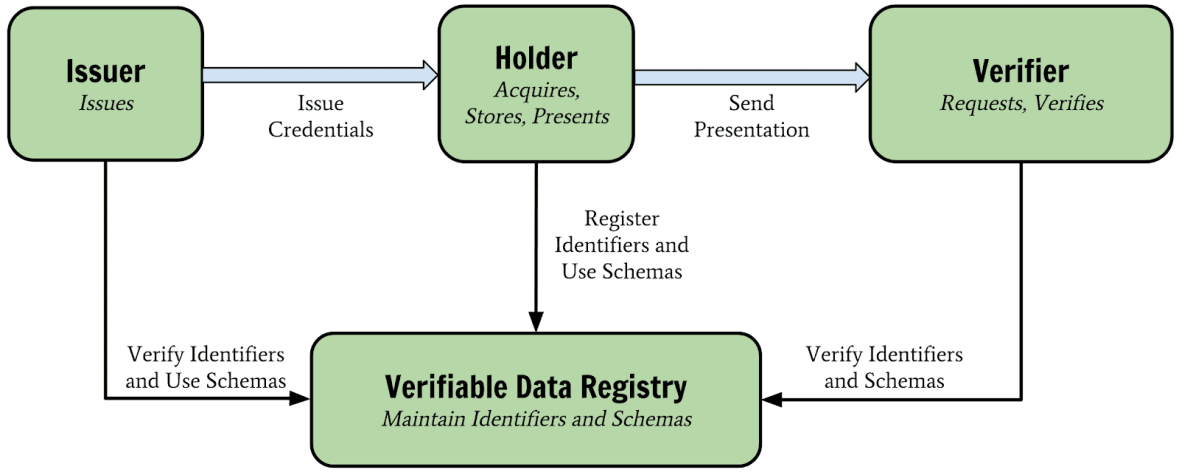
Plus,

- You can prove one more thing—that the issued credential has not been revoked.

As we'll see, verifiable credentials and presentations are not simple documents that anyone can create and use. They are cryptographically constructed so that a presentation proves the four key attributes of all credentials:

- Who issued the credential.
- The credential was issued to the entity presenting it.
- The claims were not tampered with.
- The credential has not been revoked.

Unlike a paper credential, those four attributes are evaluated not based on the judgment and expertise of the person looking at the credential, but rather online using cryptographic algorithms that are extremely difficult to forge. When a verifier receives a presentation from a holder, they use information from a blockchain (shown as the **verifiable data registry** in the image below) to perform the cryptographic calculations necessary to prove the four attributes. Forgeries become much (MUCH!) harder with verifiable credentials!



The W3C Verifiable Credentials Model
Licensed under CC BY 4.0

The prerequisite course, *LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa*, more than covers the reasons why we need a better identity model on the Internet so we won't go into it too much here. Suffice to say, blockchain has enabled a better way to build solutions and will enable a more trusted Internet.

Key Concepts

Let's review other key concepts that you'll need for this course, such as:

- self-sovereign identity
- trust over IP
- decentralized identifiers
- zero-knowledge proof
- selective disclosure
- wallet
- agent

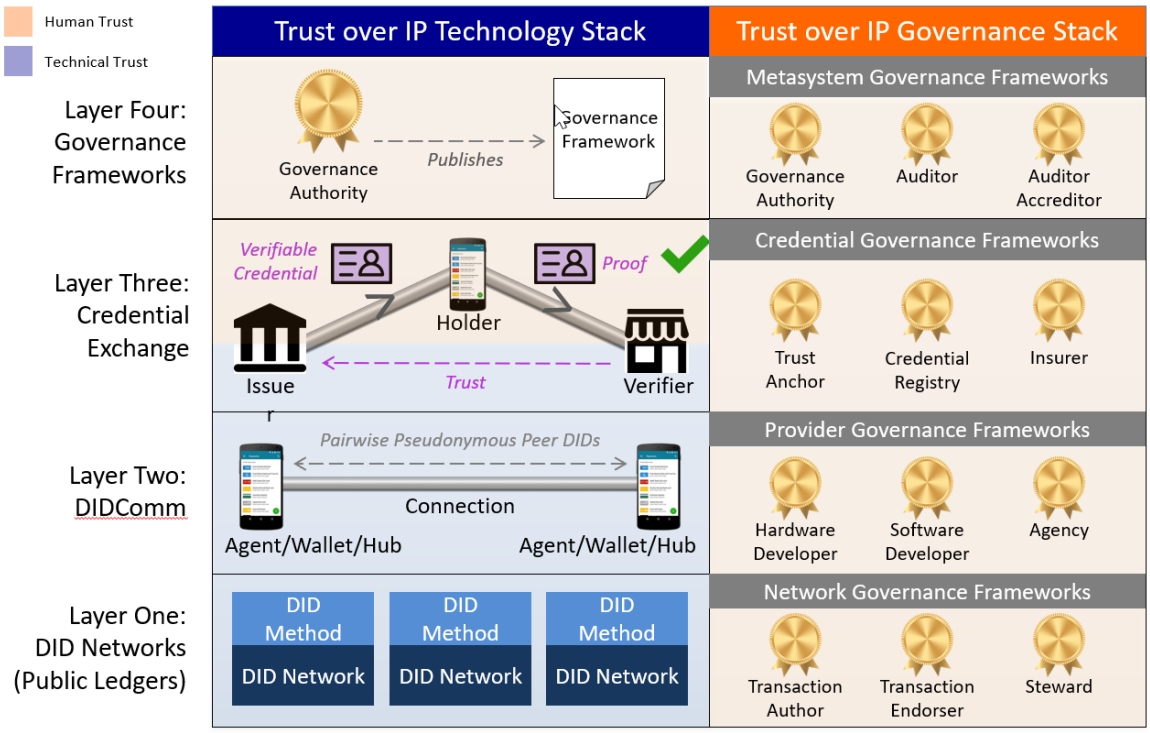
If you are not familiar or comfortable with these concepts and terminology, we suggest you refresh yourself with this course: *LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa*.

Self-Sovereign Identity (SSI)

Self-sovereign identity is one of the most important concepts discussed in the prerequisite course and it is what you should keep in mind at all times as you dig deep into the Aries world of development. SSI is the idea that you control your own data and you control when and how it is provided to others; when it is shared, it is done so in a trusted way. With SSI, there is no central authority holding your data that passes it on to others upon request. And because of the underlying cryptography and blockchain technology, SSI means that you can present claims about your identity and others can verify it with cryptographic certainty.

Trust Over IP (ToIP)

Along with SSI, another term you will hear is **trust over IP** (ToIP). ToIP is a set of protocols being developed to enable a layer of trust on the Internet, protocols embodied in Indy, Aries and Ursa. It includes self-sovereign identity in that it covers identity, but goes beyond that to cover any type of authentic data. Authentic data in this context is data that is not necessarily a credential (attributes about an entity) but is managed as a credential and offers the same guarantees when proven. ToIP is defined by the "Trust over IP Technology Stack," as represented in this image from Drummond Reed:



Trust Over IP (ToIP) Technology Stack
Licensed under CC BY 4.0

The core of Aries implements Layer Two (DIDComm) of the ToIP stack, enabling both Layer One (DIDs) and Layer Three (Credential Exchange) capabilities. Hyperledger Indy provides both a DID method and a verifiable credential exchange model. Aries is intended to work with Indy and other Layer One and Layer Three implementations. Aries implements DIDComm, a secure, transport-agnostic mechanism for sending messages based on DIDs.

Unit

A decentralized identifier is like a universally unique identifier (uuid) for your identity. DIDs are 128-bit numbers written in Base58:

did:sov:AKRMugEbG3ez24K2xnqqrm

Some things you should know about DIDs:

- A DID is controlled by one or more Ed25519 public/private key pairs. A public key is called a “verkey” (verification key); private key is called a “signing key.”
- DIDs can be created on many different blockchains; right now, Indy only supports Sovrin-style DIDs.

The DID specification can be found in the W3C Working Draft, Decentralized Identifiers (DIDs) v1.0.

Zero-Knowledge Proof

A **zero-knowledge proof** (ZKP) is about proving attributes about an entity (a person, organization or thing) without exposing a correlatable identifier about that entity. Formally, it's about presenting claims from verifiable credentials without exposing the key (and hence a unique identifier) of the proving party to the verifier. A ZKP still exposes the data asked for (which could uniquely identify the prover), but does so in a way that shows the prover possesses the issued verifiable credential while preventing multiple verifiers from correlating the prover's identity. Indy, based on underlying Ursa cryptography, implements ZKP support.

Selective Disclosure

The Indy ZKP model enables some additional capabilities beyond most non-ZKP implementations. Specifically, that claims from verifiable credentials can be **selectively disclosed**, meaning that just some data elements from credentials, even across credentials can (and should be) provided in a single presentation. By providing them in a single presentation, the verifier knows that all of the credentials were issued to the same entity. In addition, a piece of cryptography magic that is part of ZKP (that we won't detail here—but it's fascinating) allows proving pieces of information without presenting the underlying data. For example, proving a person is older than a certain age based on their date of birth, but without providing their date of birth to the verifier. Very cool!

An Example of Selective Disclosure
Person is verified as "old enough" to enter the bar but no other data is revealed
By Peter Stokyo
Licensed under CC BY 4.0

Wallet

DIDs and their keys are stored in an identity **wallet**. Identity wallets are like cryptocurrency wallets, but store different kinds of data—DIDs, credentials and metadata to use them. You can find more info on GitHub. The **indy-sdk** includes a default implementation of a wallet

that works out of the box. A wallet is the software that processes verifiable credentials and DIDs.

Agent

Indy, Aries and Ursa use the term **agent** to mean the software that interacts with other entities (via DIDs and more). For example, a person might have a mobile agent app on their smart device, while an organization might have an enterprise agent running on an enterprise server, perhaps in the cloud. All agents (with rare exceptions) have secure storage for securing identity-related data including DIDs, keys and verifiable credentials. As well, all agents are controlled to some degree by their owner, sometimes with direct control (and comparable high trust) and sometimes with minimal control, and far less trust.

Summary

This chapter has largely been a review of the concepts introduced in the previous course. Its purpose is to provide context for why you want to become an Aries developer and recaps some of the terminology and concepts behind decentralized identity solutions that were discussed in the prerequisite course: *LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa*.

Chapter 2: Exploring Aries and Aries Agents

Chapter Overview

As you learned in the prerequisite course (*LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa*)—which you took, right?—Hyperledger Aries is a toolkit for building solutions focused on creating, transmitting, storing and using verifiable credentials. Aries agents are software components that act on behalf of ("as agents of") entities—people, organizations and things. They enable decentralized, self-sovereign identity based on a secure, peer-to-peer communications channel. In fact, the main reason for using an Aries agent is to exchange verifiable credentials!

In this chapter, we'll look at the architecture of an Aries agent. Specifically, what parts of an agent come from Aries, and what parts you are going to have to build. We will also look at the the interfaces that exist to allow Aries agents to talk to one another and to public ledgers such as instances of Hyperledger Indy.

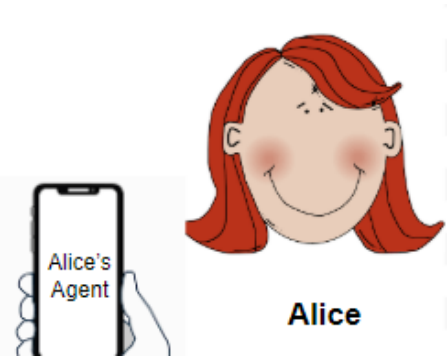
Learning Objectives

By the end of this chapter you should:

- Be familiar with the Aries ecosystem consisting of edge agents for people and organizations, cloud agents for enterprises and device agents for IoT devices.
- Know the concepts behind issuing and verifying agents.
- Understand the internal components of an Aries agent.

Examples of Aries Agents

Let's first look at a couple of examples to remind us what Aries agents can do. We'll also use this as a chance to introduce some characters that the Aries community has adopted for many Aries proof-of-concept implementations. You first met these characters in the LFS172x course.



- Alice is a person who has a mobile Aries agent on her smartphone. She uses it to receive credentials from various entities, and uses those credentials to prove things about herself online.
- Alice's smartphone app connects with a cloud-based Aries agent that routes messages to her. It too is Alice's agent, but it's one that is (most likely) run by a vendor. We'll learn more about these cloud services when we get to the Aries mobile agents chapter.
- Alice is a graduate of Faber College (of Animal House fame), where the school slogan is "Knowledge is Good." (We think the slogan should really be "Zero Knowledge is Good.") Faber College has an Aries agent that issues verifiable credentials to the college's students.

- Faber also has agents that verify presentations of claims from students and staff at the college to enable access to resources. For example, Alice proves the claims from her “I graduated from Faber College” credential to get a discount at the Faber College Bookstore whenever she is on campus—or when she shops there online.
- Faber also has an Aries agent that receives, holds and proves claims from credentials about the college itself. For example, Faber’s agent might hold a credential that it is authorized to grant degrees to its graduates from the jurisdiction (perhaps the US state) in which it is registered.
- ACME is a company for whom Alice wants to work. As part of their application process, ACME’s Aries agent requests proof of Alice’s educational qualifications. Alice’s Aries agent can provide proof using the credential issued by Faber to Alice.
- Since Alice is the first Faber College student to ever apply to ACME, ACME doesn’t know if they can trust Faber. An ACME agent might connect to Faber’s agent (using the DID in the verifiable credential that Alice presented) to get proof that Faber is a credentialed academic institution.

Lab: Issuing, Holding, Proving and Verifying

In this first lab, we’ll walk through the interactions of three Aries agents:

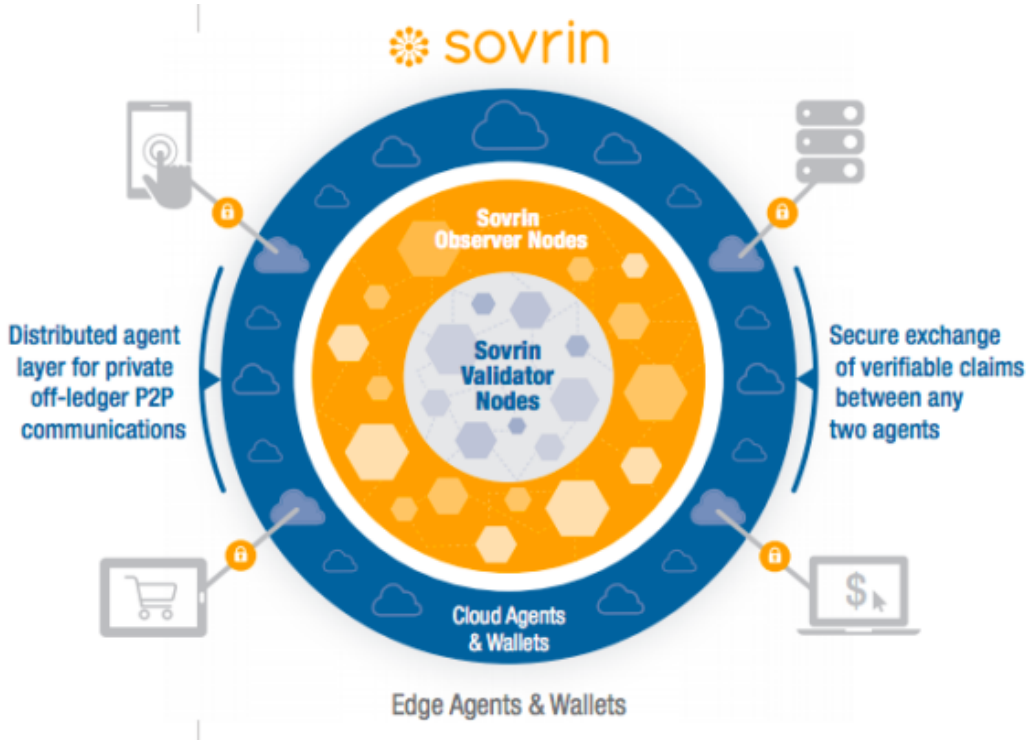
- A mobile agent to hold a credential and prove claims from that credential.
- An issuing agent.
- A verifying agent.

The instructions for running the lab can be found on GitHub.

An Aries Ecosystem

All of the examples of Aries agents in the previous section can be built independently by different organizations because of the common protocols upon which Aries agents communicate. This classic Indy ecosystem picture shows multiple agents—the four around the outside (on a phone, a tablet, a laptop and an enterprise server) are referred to as **edge agents**, and the ones in the blue circle are called **cloud agents**. In the centre is

the blockchain—the public ledger on which the globally resolvable data that supports verifiable credentials reside.



The Aries Ecosystem
Licensed under CC BY 4.0

The agents in the picture share many attributes:

- They all have storage for keys.
- They all have some secure storage for other data related to their role as an agent.
- Each interacts with other agents using secure, peer-to-peer messaging protocols.
- They all have an associated mechanism to provide “business rules” to control the behavior of the agent:
  – Often a person (via a user interface) for phone, tablet, laptop, etc.-based agents.
  – Often a backend enterprise system for enterprise agents.
  – For cloud agents the “rules” are often limited to the routing of messages to and from other agents.

While there can be many other agent setups, the picture above shows the most common ones:

- Edge agents for people.
- Edge agents for organizations.
- Cloud agents for routing messages between agents (although cloud agents could also be edge agents).

A significant emerging use case missing from that picture is agents embedded within or associated with IoT devices. In the common IoT case, IoT device agents are just variants of other edge agents, connected to the rest of the ecosystem through a cloud agent. All the same principles apply.

A bit misleading in the picture is the implication that edge agents connect to the public ledger through cloud agents. In fact, (almost) all agents connect directly to the ledger network. In this picture it's the Sovrin ledger, but that could be any Indy network (e.g. a set of nodes running **indy-node** software) and, in future Aries agents, ledgers from other providers. Since most agents connect directly to ledgers, most embed ledger SDKs (e.g. **indy-sdk**) and make calls to the ledger SDK to interact with the ledger and other SDK controlled resources (e.g. secure storage). Super small IoT devices are an instance of an exception to that. Lacking compute/storage resources and/or connectivity, such devices might securely communicate with a cloud agent that in turn communicates with the ledger.

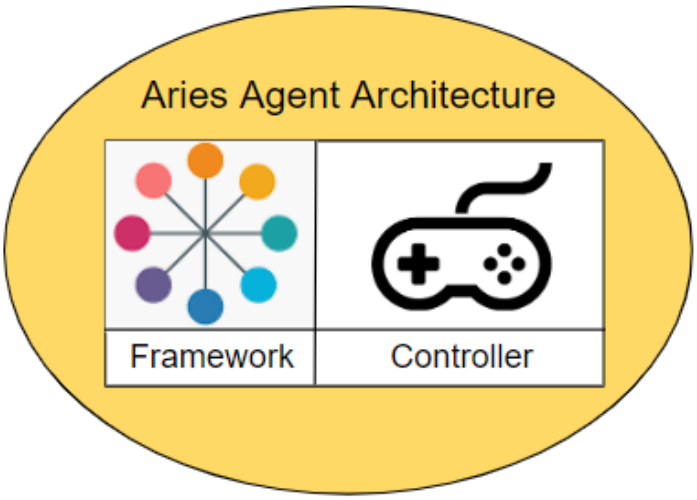
The (most common) purpose of cloud agents is to enable secure and privacy preserving routing of messages between edge agents. Rather than messages going directly from edge agent to edge agent, messages sent from edge agent to edge agent are routed through a sequence of cloud agents. Some of those cloud agents might be controlled by the sender, some by the receiver and others might be gateways owned by agent vendors (called "agencies"). In all cases, an edge agent tells routing agents "here's how to send messages to me," so a routing agent sending a message only has to know how to send a peer-to-peer message—a single hop in the message's journey. While quite complicated, the protocols used by the agents largely take care of this complexity, and most developers don't have to know much about it. We'll cover more about routing in the mobile agents chapter of this course.

Note: You may have noticed many caveats in this section: "most common," "commonly" and so on. That is because there are many small building blocks available in Aries and underlying components that can be combined in infinite ways. We recommend not worrying about the alternate use cases for now. Focus on understanding the common use cases (listed earlier) while remembering that other configurations are possible.

Note: While most Aries agents currently available only support Indy-based ledgers, the stated intention of the Aries project is to add support for other ledgers. In this course, we'll focus mostly on the Aries code that is being used today, which is mostly based on Indy, but we'll highlight where and how other ledgers can and are being integrated.

The Logical Components of an Aries Agent

All Aries agent deployments have two logical components: a **framework** and a **controller**.



The Logical Components of an Aries Agent
Licensed under CC BY 4.0

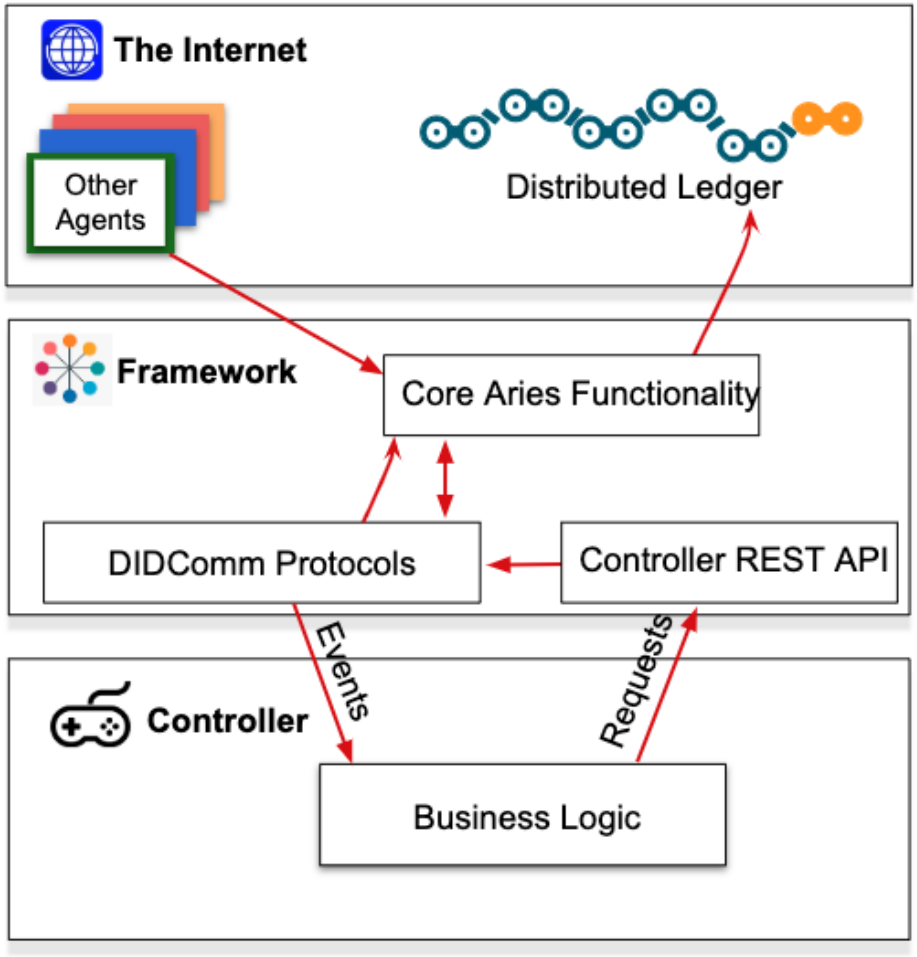
The framework contains the standard capabilities that enable an Aries agent to interact with its surroundings—ledgers, storage and other agents. A framework is an artifact of an Aries project that you don't have to create or maintain, you just embed in your solution. The framework knows how to initiate connections, respond to requests, send messages and more. However, a framework needs to be told when to initiate a connection. It doesn't know what response should be sent to a given request. It just sits there until it's told what to do.

The controller is the component that, well, controls, an instance of an Aries framework's behavior—the business rules for that particular instance of an agent. The controller is the part of a deployment that you build to create an Aries agent that handles your use case for responding to requests from other agents, and for initiating requests. For example:

- In a mobile app, the controller is the user interface and how the person interacts with the user interface. As events come in, the user interface shows the person their options, and after input from the user, tells the framework how to respond to the event.
- An issuer, such as Faber College's agent, has a controller that integrates agent capabilities (requesting proofs, verifying them and issuing credentials) with enterprise systems, such as a Student Information System that tracks students and their grades. When Faber's agent is interacting with Alice's, and Alice's requests an "I am a Faber Graduate" credential, it's the controller that figures out if Alice has earned such a credential, and if so, what claims should be put into the credential. The controller also directs the agent to issue the credential.

Aries Agent Architecture (ACA-PY)

The diagram below is an example of an Aries agent architecture, as exemplified by Aries Cloud Agent – Python (ACA-Py):



Aries Agent Architecture (ACA-PY)

The framework provides all of the core Aries functionality such as interacting with other agents and the ledger, managing secure storage, sending event notifications to, and receiving instructions from the controller. The controller executes the business logic that defines how that particular agent instance behaves—how it responds to the events it receives, and when to initiate events. The controller might be a web or native user interface for a person or it might be coded business rules driven by an enterprise system.

Between the two is a pair of notification interfaces.

- When the framework receives a message (an event) from the outside world, it sends a notification about the event to the controller so the controller can decide what to do.
- In turn, the controller sends a notification to the framework to tell the framework how to respond to the event.
  – The same controller-to-framework notification interface is used when the controller wants the framework to initiate an action, such as sending a message to another agent.

What that means for an Aries developer is that the framework you use is a complete dependency that you include in your application. You don't have to build it yourself. It is the controller that gives your agent its unique personality. Thus, the vast majority of Aries developers focus on building controllers. Of course, since Aries frameworks are both evolving and open source, if your agent implementation needs a feature that is not in the framework you are building, you can do some Aries framework development and contribute it to Hyperledger.

Agent Terminology Confusion

In many places in the Aries community, the "agent framework" term we are using here is shortened to "agent." That creates some confusion as you now can say "an Aries agent consists of an agent and a controller." Uggghh… Throughout the course we have tried to make it very clear when we are talking about the whole agent versus just the agent framework. Often we will use the name of a framework to make it clear the context of the term. However, as a developer, you should be aware that in the community, the term "agent" is sometimes used just for the agent framework component and sometimes for the combined framework+controller.

Naming is hard… 🤦‍♂️

Current Agent Frameworks

There are several Aries general purpose agent frameworks that are ready to go out of the box. The links to the repos are embedded.

- **aries-cloudagent-python** (ACA-Py) is suitable for all non-mobile agent applications and has production deployments. ACA-Py and a controller run together, communicating across an HTTP interface. Your controller can be written in any language and ACA-Py embeds the **indy-sdk**. At the time of writing this course, ACA-Py does not support any other ledgers or verifiable credential exchange models other than Hyperledger Indy.
- **aries-framework-dotnet** can be used for building mobile (via Xamarin) and server-side agents and has production deployments. The controller for an **aries-framework-dotnet** app can be written in any language that supports embedding the framework as a library in the controller. The framework embeds the **indy-sdk**.

- **aries-ataticagent-python** is a configurable agent that does not use persistent storage. To use it, keys are pre-configured and loaded into the agent at initialization time.

There are several other frameworks that are currently under active development, including the following.

- **aries-framework-go** is a pure golang framework that provides a similar architecture to ACA-Py, exposing an HTTP interface for its companion controller. The framework does not currently embed the Indy SDK and work on supporting a golang-based verifiable credentials implementation is in progress.
- **aries-sdk-ruby** is a Ruby-on-Rails agent framework with a comparable architecture to ACA-Py.
- **aries-framework-javascript** is a pure JavaScript framework that is intended to be the basis of React Native mobile agent implementations.

Different Aries frameworks implement the interfaces between the framework and controller differently. For example, in **aries-cloudagent-python**, the agent framework and the controller use HTTP interfaces to interact. In **aries-framework-dotnet**, the framework is a library, and the interfaces are within the process as calls and callbacks to/from the library.

Aries Agent Internals and Protocols

In this section, we'll cover, at a high level, the internals of Aries agents and how Aries agent messaging protocols are handled.

The most basic function of an Aries agent is to enable (on behalf of its controlling entity) secure messaging with other agents. Here's an overview of how that happens:

- Alice and Bob have running agents.
- Somehow (we'll get to that) Alice's agent discovers Bob's agent and sends it an invitation (yes, we'll get to that too!) to connect.
  – The invitation is in plaintext (perhaps presented as a QR code) and includes information on how a message can be securely sent back to Alice.
- Bob's agent (after conferring with Bob—"Do you want to do this?") creates a private DID for the relationship and embeds that in a message to Alice's agent with a request to connect.
  – This message uses information in the invitation to securely send the message back to Alice's agent.
- Alice's agent associates the message from Bob's agent with the invitation that was sent to Bob and confers with Alice about what to do.
- Assuming Alice agrees, her agent stores Bob's connection information, creates a private DID for the relationship, and sends a response message to Bob's agent.
  – Whoohoo! The agents are connected.
- Using her agent, Alice can now send a message to Bob (via their agents, of course) and vice-versa. The messages use the newly established

communication channel and so are both private and secure.

Lab: Agents Connecting

Let's run through a live code example of two agents connecting and messaging one another. Follow this link to try it yourself and explore the code in the example.

Summary

This chapter focused on the Aries ecosystem (the way Aries is used in the real world), the Aries agent architecture (the components that make up an Aries agent), and how an Aries agent functions. We looked at examples of Aries agents, namely Alice and Faber College, and you stepped through a demo to verify, hold and issue verifiable credentials. We learned the difference between an edge agent and a cloud agent and common use cases for each. Next, we described the Aries agent architecture, discussing an agent framework, its controller and how the two work together. The main takeaway from this chapter is that as a developer, you will most likely be building your own controller, which will give your agent the business rules it needs to follow depending on your agent implementation.

You'll notice in the example provided on the *Aries Agents Internals and Protocols* page, there was no mention of a public ledger. That's right, Aries agents can provide a messaging interface without any ledger at all! Of course, since the main reason for using an Aries agent is to exchange verifiable credentials, and verifiable credential exchange requires a ledger, we'll look at ledgers in the next section.

Chapter 3: Running a Network for Aries Development

Chapter Overview

In the last chapter, we learned all about the Aries agent architecture or the internals of an agent: the controller and framework. We also discussed common setups of agents and some basics about messaging protocols used for peer-to-peer communication. The lab in Chapter 2 demonstrated connecting two agents that didn't use a ledger. Now that you're comfortable with what an agent does and how it does it—and have seen agents at work off-ledger—let's set the groundwork for using a ledger for your development needs.

Learning Objectives

In this chapter, we will describe:

- What you need to know and *not know* about ledgers in order to build an SSI application.
- How to get a local Indy network up and running, plus other options for getting started.
- The ledger's genesis file—the file that contains the information necessary for an agent to connect to that ledger.

Ledgers: What You Don't Need To Know

Many people come to the Indy and Aries communities thinking that because the projects are "based on blockchain," that the most important thing to learn first is about the underlying blockchain. Even though their goal is to build an application for their use case, they dive into Indy, finding the guide on starting an Indy network and off they go—bumping their heads a few times on the way. Later, when they discover Aries and what they really need to do (build a controller, which is really, just an app with APIs), they discover they've wasted a lot of time.

Don't get us wrong. The ledger is important, and the attributes (robustness, decentralized control, transaction speed) are all key factors in making sure that the ledger your application will use is sufficient. It's just that as an Aries agent application developer, the details of the ledger are *Someone Else's Problem*. There are three situations in which an organization producing self-sovereign identity solutions will need to know about ledgers:

- If your organization is going to operate a ledger node (for example, a steward on the Sovrin network), the operations team needs how to install and maintain that node. There is no development involved in that work, just operation of a product.
- If you are a developer that is going to contribute code to a ledger project (such as Indy) or interface to a ledger not yet supported by Aries, you need to know about the development details of that ledger.
- If you are building a product for a specific use case, the business team must select a ledger that is capable of supporting that use case. Although there is some technical knowledge required for that, there is no developer knowledge needed.

So, assuming you are here because you are building an application, the less time you spend on ledgers, the sooner you can get to work on developing an application. For now, skip diving into Indy and use the tools and techniques outlined here. We'll also cover some additional details about integrating with ledgers in Chapter 8, *Planning for Production*, later in this course.

With that, we'll get on with the minimum you have to know about ledgers to get started with Aries development. In the following, we assume you are building an application for running against a Hyperledger Indy ledger.

Running a Local Indy Network

The easiest way to get a local Indy network running is to use von-network, a pre-packaged Indy network built by the Government of British Columbia's (BC) VON team. In addition to providing a way to run a minimal four-node Indy network using docker containers with just two commands, von-network includes:

- A well maintained set of versioned Indy container images.
- A web interface allowing you to browse the transactions on the ledger.
- An API for accessing the network's genesis file (see below).

- A web form for registering DIDs on the network.
- Guidance for running a network in a cloud service such as Amazon Web Service or Digital Ocean.

The VON container images maintained by the BC Gov team are updated with each release of Indy, saving others the trouble of having to do that packaging themselves. A lab focused on running a VON network instance will be provided at the end of this chapter.

Or, Don't Run a Network for Aries Development

What is easier than running a local network with von-network? How about not running a local network at all.

Another way to do development with Indy is to use a public Indy network sandbox. With a sandbox network, each developer doesn't run their own local network, they access one that is running remotely. With this approach, you can run your own, or even easier, use the BC Government's BCovrin (pronounced "Be Sovereign") networks (dev and test). As of writing this course, the networks are open to anyone to use and are pretty reliable (although no guarantees!). They are rarely reset, so even long lasting tests can use a BCovrin network.

An important thing that a developer needs to know about using a public sandbox network is to make sure you create unique credential definitions on every run by making sure issuer DIDs are different on every run. To get into the weeds a little:

- Indy DIDs are created from a seed, an arbitrary 32-character string. A given seed passed to Indy for creating a DID will always return the same DID, public key and private key.
- Historically, Indy/Aries developers have configured their apps to use the same seeds in development so that the resulting DIDs are the same every time. This works if you restart (delete and start fresh) the ledger and your agent storage on every run, but causes problems when using a long lasting ledger.
  – Specifically, a duplicate credential definition (same DID, name and version) to one already on a ledger will fail to be written to the ledger.
- The best solution is to configure your app so a randomly generated seed is used in development such that the issuer's DID is unique on every run so that the credential definition name and version can remain the same on every run.

Note: This is important for development. We'll talk about some issues related to going to production in Chapter 8, *Planning for Production*, where the problem is reversed—we MUST use the same DID and credential definition every time we start an issuer agent.

In the labs in this course, you will see examples of development agents running against both local and remote sandbox Indy networks.

Proof of Concept Networks

When you get to the point of releasing a proof of concept (PoC) application “into the wild” for multiple users to try, you will want to use an Indy network that all of your PoC participants can access. As well, you will want that environment to be stable such that it is always available when it’s needed. We all know about how mean the Demo Gods can be!

Some of the factors related to production applications (covered in Chapter 8, *Planning for Production*) will be important for a PoC. For such a test, a locally running network is not viable and you must run a publicly accessible network. For that, you have three choices:

- The BCovrin sandbox test network is available for such long term testing.
- The Sovrin Foundation, operates two non-production networks:
  – Builder Net: For active development of your solution.
  – Staging Net: For proofs of concept, demos, and other non-production but stable use cases.
  Note: Non-production Sovrin networks are permissioned, which means that you have to do a bit more to use those. We’ll cover a bit in the next section of this chapter and in Chapter 8, *Planning for Production* about getting ready for production deployments.
- You may choose to run your own network on something like Amazon Web Service or Azure. Basically, you run your own version of “BCoverin”.

Running your own network gives you the most control (and is pretty easy if you use von-network), so that might be the preferred option. However, if you need to interoperate with agents from other vendors, the public ledger you choose must be one that is supported by all agents. That often makes either or both of the BCovrin and Sovrin Foundation networks as the best, and perhaps only, choice.

Audit Access Expires Apr 24, 2020 You lose all access to this course, including your progress, on Apr 24, 2020. Upgrade by Mar 27, 2020 to get unlimited access to the course as long as it exists on the site. Upgrade nowto retain access past Apr 24, 2020 The Indy Genesis File

In working with an Indy network, the ledger’s **genesis file** contains the information necessary for an agent to connect to that ledger. Developers new to Indy, particularly those that try to run their own network, often have trouble with the genesis file, so we’ll cover it here.

The genesis file contains information about the physical endpoints (IP addresses and ports) for some or all of the nodes in the ledger pool, and the cryptographic material necessary to communicate with those nodes. Each genesis file is unique to its associated ledger, and must be available to an agent that wants to connect to the ledger. It is called the genesis file because it has information about the genesis (first) transactions on the

ledger. Recall that a core concept of blockchain is that the blocks of the chain are cryptographically tied to all the blocks that came before it, right back to the first (genesis) block on the chain.

The genesis file for Indy **sandbox** ledgers is (usually) identical, with the exception of the endpoints for the ledger—the endpoints must match where the nodes of the ledger are physically running. The cryptographic material is the same for sandbox ledgers because the genesis transactions are all the same. Those transactions:

- Create a trustee **endorser** DID on the ledger that has full write permission on the ledger.
- Permission the nodes of the ledger to process transactions.

Thus, if you get the genesis file for a ledger and you know the "magic" seed for the DID of the **trustee** (the identity owner entrusted with specific identity control responsibilities by another identity owner or with specific governance responsibilities by a governance framework), you can access and write to that ledger. That's great for development and it makes deploying proof-of-concepts easy. Configurations like von-network take advantage of that consistency, using the "magic" seed to bootstrap the agent. For agents that need to write to the network (at least credential issuers, and possibly others), there is usually an agent provisioning step where the endorser DID is used to write a DID for the issuer that has sufficient write capabilities to do whatever else it needs to do. This is possible because the seed used to create the endorser DID is well known. In case you are wondering, the magic seed is:

000000000000000000000000Trustee1

For information about this, see this great answer on Stack Overflow about where it comes from.

As we'll see in Chapter 8, *Preparing for Production*, the steps are conceptually the same when you go to production—use a **transaction endorser** that has network write permissions to create your DID. However, you'll need to take additional steps when using a production ledger (such as the Sovrin Foundation's Main Net) because you won't know the seed of any endorsers on the network. Connecting to a production ledger is just as easy—you get the network's genesis file and pass that to your agent. However, being able to write to the network is more complicated because you don't know the "magic DID" that enables full write access.

By the way, the typical problems that developers have with genesis files is either they try to run an agent without a genesis file, or they use a default genesis file that has not had the endpoints updated to match the location of the nodes in your network.

Genesis File Handling in Aries Frameworks

Most Aries frameworks make it easy to pass to the agent the genesis file for the network to which it will connect. For example, we'll see from the labs in the next chapter that to

connect an instance of an ACA-Py agent to a ledger you use command line parameters to specify either a file name for a local copy of the genesis file, or a URL that can is resolved to fetch the genesis file. The latter is often used by developers that use the VON Network because each VON Network instance has a web server deployed with the network and provides the URL for the network's genesis file. The file is always generated after deployment, so the endpoints in the file are always accurate for that network.

Lab: Running a VON Network Instance

Please follow this link to run a lab in which you will start a VON Network, browse the ledger, look at the genesis file and create a DID.

Summary

The main point of this chapter is to get you started in the right spot: you don't need to dig deep into the ledger in order to develop SSI applications. You should now be aware of the options running an Indy network and know the importance of the genesis file for your particular network.

In the last chapter, we covered the architecture of an agent and demonstrated connecting two agents that didn't use a ledger. In this chapter, we covered running a ledger. So, in the next chapter, let's combine the two and look at running agents that connect to a ledger.

Chapter 4: Developing Aries Controllers

Chapter Overview

In the second chapter, we ran two simple command line agents that connected and communicated with one another. In the third chapter, we went over running a local ledger. In this chapter, we'll go into details about how you can build a controller by running agents that connect, communicate and use a ledger to exchange credentials. For developers wanting to build applications on top of Aries and Indy, this chapter is the core of what you need to know and is presented mostly as a series of labs.

We will learn about controllers by looking at **aries-cloudagent-python** (ACA-Py) in all the examples. Don't worry, with ACA-Py, a controller (as we will see) can be written in any language, so if you aren't a Python-er, you'll be fine. However, ACA-Py is not suitable for mobile so we'll leave that discussion until Chapter 7. In the last section of this chapter, we'll talk briefly about the other current Aries open source frameworks.

Learning Objectives

In this chapter, you will learn:

- What an agent needs to know at startup.
- How protocols impact the structure and role of controllers.

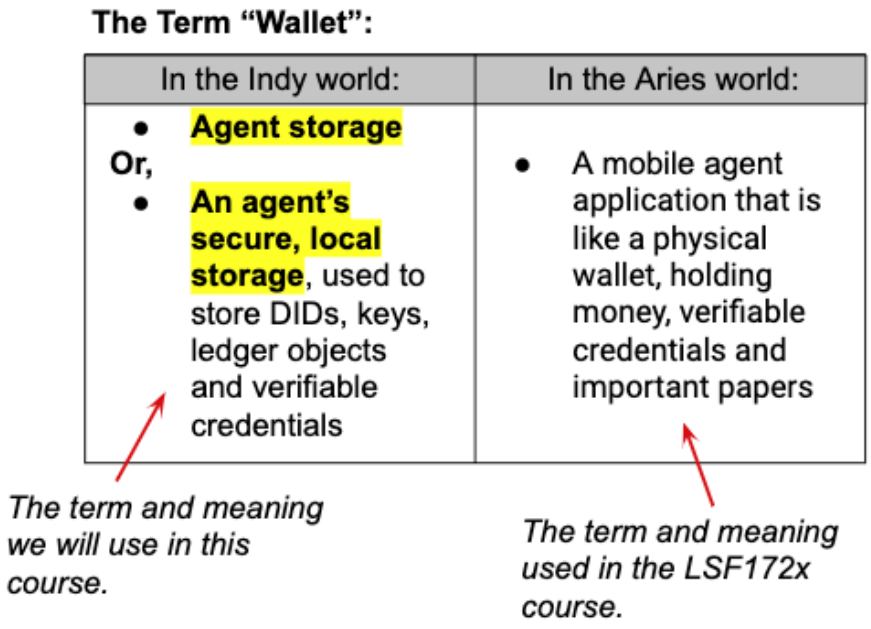
- About the **aries-cloudagent-python** (ACA-Py) framework versus other Aries frameworks (such as **aries-framework-dotnet** and **aries-framework-go**).

The goal of this chapter is to have you build your own controller using the framework of your choice.

Aside: The Term “Wallet”

In the prerequisite course, *LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa*, we talked about the term “wallet” as a mobile application that is like a physical wallet, holding money, verifiable credentials and important papers. This is the definition the Aries community would like to use. Unfortunately, the term has historically had a second meaning in the Indy community, and it’s a term that developers in Aries and Indy still see. In Indy, the term “wallet” is used for the storage part of an Indy agent, the place in the agent where DIDs, keys, ledger objects and credentials reside. That definition is still in common use in the source code that developers see in Indy and Aries.

So, while the Indy use of the term wallet is being eliminated in Aries, because of the use of the existing Indy components in current Aries code (at least at the time of writing this course), we’re going to be using the Indy meaning for the term wallet—an agent’s secure, local storage. We’ll also use the term “agent storage” to mean the same thing.



“Wallet” in the Indy and Aries Worlds

Agent Start Up

An Aries agent needs to know a lot of configuration information as it starts up. For example, it needs to know:

- The location of the genesis file for the ledger it will use (if any).

- If it needs to create items (DIDs, schema, etc.) on the ledger, how to do that, or how to find them if they have already been created.
- Transport (such as HTTP or web sockets) endpoints for messaging other agents.
- Storage options for keys and other data.
- Interface details between the agent framework and the controller for events and requests.

These are concepts we've talked about earlier and that you should recognize. Need a reminder? Checkout the "*Aries Agent Architecture*" section of Chapter 2. We'll talk about these and other configuration items in this chapter and the next.

Command Line Parameters

Most of the options for an agent are configured in starting up ACA-Py using command line options in the form of "**–option <extra info>**". For example, to specify the name of the genesis file the agent is to use, the command line option is "**–genesis-file <genesis-file>**". The number of startup options and the frequency with which new options are added are such that ACA-Py is self-documenting. A "**–help**" option is used to get the list of options for the version of ACA-Py you are using.

The provision and start Options

An agent is a stateful component that persists data in its wallet and to the ledger. When an agent starts up for the very first time, it has no persistent storage and so it must create a wallet and any ledger objects it will need to fulfill its role. When we're developing an agent, we'll do that over and over: start an agent, create its state, test it, stop it and delete it. However, when an agent is put into production, we only initialize its state once. We must be able to stop and restart it such that it finds its existing state, without having to recreate its wallet and all its contents from scratch.

Because of this requirement of a one time "start from scratch" and a many times "start with data," ACA-Py provides two major modes of operation, **provision** and **start**. **Provision** is intended to be used one time per agent instance to establish a wallet and the required ledger objects. This mode may also be used later when something new needs to be added to the wallet and ledger, such as an issuer deciding to add a new type of credential they will be issuing. **Start** is used for normal operations and assumes that everything is in place in the wallet and ledger. If not, it should error and stop—an indicator that something is wrong.

The **provision** and **start** separation is done for security and ledger management reasons. Provisioning a new wallet often (depending on the technical environment) requires higher authority (e.g. root) database credentials. Likewise, creating objects on a ledger often requires the use of a DID with more access permissions. By separating out provisioning from normal operations, those higher authority credentials do not need to be available on an ongoing basis. As well, on a production ledger such as Sovrin, there is a cost to write to

the ledger. You don't want to be accidentally writing ledger objects as you scale up and down ACA-Py instances based on load. We've seen instances of that.

We recommend the pattern of having separate provisioning and operational applications, with the operational app treating the ledger as read-only. When initializing the operational app, it should verify that all the needed objects are available in the wallet and ledger, but should error and stop if they don't exist. During development, we recommend using a script to make it easy to run the two steps in sequence whenever they start an environment from scratch (a fresh ledger and empty wallet).

The only possible exception to the "no writes in start mode" method is the handling of credential revocations, which involve ledger writes. However, that's pretty deep into the weeds of credential management, so we won't go further with that here.

Startup Option Groups

The ACA-Py startup options are divided into a number of groups, as outlined in the following:

- **Debug**: Options for development and debugging. Most (those prefixed with "auto-") implement default controller behaviors so the agent can run without a separate controller. Several options enable extra logging around specific events, such as when establishing a connection with another agent.
- **Admin**: Options to configure the connection between ACA-Py and its controller, such as on what endpoint and port the controller should send requests. Important required parameters include if and how the ACA-Py/controller interface is secured.
- **General**: Options about extensions (external Python modules) that can be added to an ACA-Py instance and where non-Indy objects are stored (such as connections and protocol state objects).
- **Ledger**: Options that provide different ways for the agent to connect to a ledger.
- **Logging**: Options for setting the level of logging for the agent and to where the logging information is to be stored.
- **Protocol**: Options for special handling of several of the core protocols. We'll be going into a deeper discussion of protocols in the next chapter.
- **Transport**: Options about the interfaces that are to be used for connections and messaging with other agents.
- **Wallet**: Options related to the storage of keys, DIDs, Indy ledger objects and credentials. This includes the type of database (e.g. SQLite or PostgreSQL) and credentials for accessing the database.

Note: While the naming and activation method of the options are specific to ACA-Py, few are exclusive to ACA-Py. Any agent, even those from other Aries frameworks, likely offer (or should offer) these options.

Lab: Agent Startup Options

Here is a short lab to show you how you can see all of the ACA-Py startup options.

The many ACA-Py startup options can be overwhelming. We'll address that in this course by pointing out which options are used by the different sample agents with which we'll be working.

How Aries Protocols Impact Controllers

Before we get into the internals of controllers, we need to talk about Aries protocols, introduced in Chapter 5 of the prerequisite course (*LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa*) and a subject we'll cover much deeper in the next chapter. For now, we'll cover just enough to understand how protocols impact the structure and role of controllers.

As noted earlier, Aries agents communicate with each other via a message mechanism called DIDComm (DID Communication). DIDComm uses DIDs (specifically private, pairwise DIDs—usually) to enable secure, asynchronous, end-to-end encrypted messaging between agents, with messages (usually) routed through some configuration of intermediary agents. Aries agents use (an early instance of) the did:peer DID method, which uses DIDs that are not published to a public ledger, but that are only shared privately between the communicating parties, usually just two agents.

The caveats in the above paragraph:

- An enterprise agent may use a public DID for all of its peer-to-peer messaging.
- Agents may directly message one another without any intermediate agents.
- The early version of did:peer does not support rotating keys of the DID.

Given the underlying secure messaging layer (routing and encryption are covered Chapter 7), Aries protocols are standard sequences of messages communicated on the messaging layer to accomplish a task. For example:

- The **connection protocol** (RFC 0160) enables two agents to establish a connection through a series of messages—an invitation, a connection request and a connection response.
- The **issue credential protocol** enables an agent to issue a credential to another agent.
- The **present proof protocol** enables an agent to request and receive a proof from another agent.

Each protocol has a specification that defines the protocol's messages, a series of named states, one or more roles for the different participants, and a state machine that defines the state transitions triggered by the messages. For example, the following table shows the messages, roles and states for the connection protocol. Each participant in an instance of a protocol tracks the state based on the messages they've seen. Note that the states of the agents executing a protocol may not be the same at any given time. An agent's state

depends on the most recent message received or sent by/to that agent. The details of the components of all protocols are covered in Aries RFC 0003 (RFC 0003).

Message	Role of Sender	State
invitation	**inviter**	**invited**
connectionRequest	**invitee**	**requested**
connectionResponse	**inviter**	**connected**

In ACA-Py, the code for protocols is implemented as (possibly externalized) Python modules. All of the core modules are in the ACA-Py code base itself. External modules are not in the ACA-Py code base, but are written in such a way that they can be loaded by an ACA-Py instance at run time, allowing organizations to extend ACA-Py (and Aries) with their own protocols. Protocols implemented as external modules can be included (or not) in any given agent deployment. All protocol modules include:

- The definition of a state object for the protocol.
  – The protocol state object gets saved in agent storage while an instance of a protocol is waiting for the next event.
- The handlers for the protocol messages.
- The events sent to the controller on receipt of messages that are part of the protocol.
- Administrative messages that are available to the controller to inject business logic into the running of the protocol.

Each administrative message (endpoint) becomes part of the HTTP API exposed by the agent instance, and is visible by the OpenAPI/Swagger generated user interface. More on that later in this chapter.

Aries Protocols: The Controller Perspective

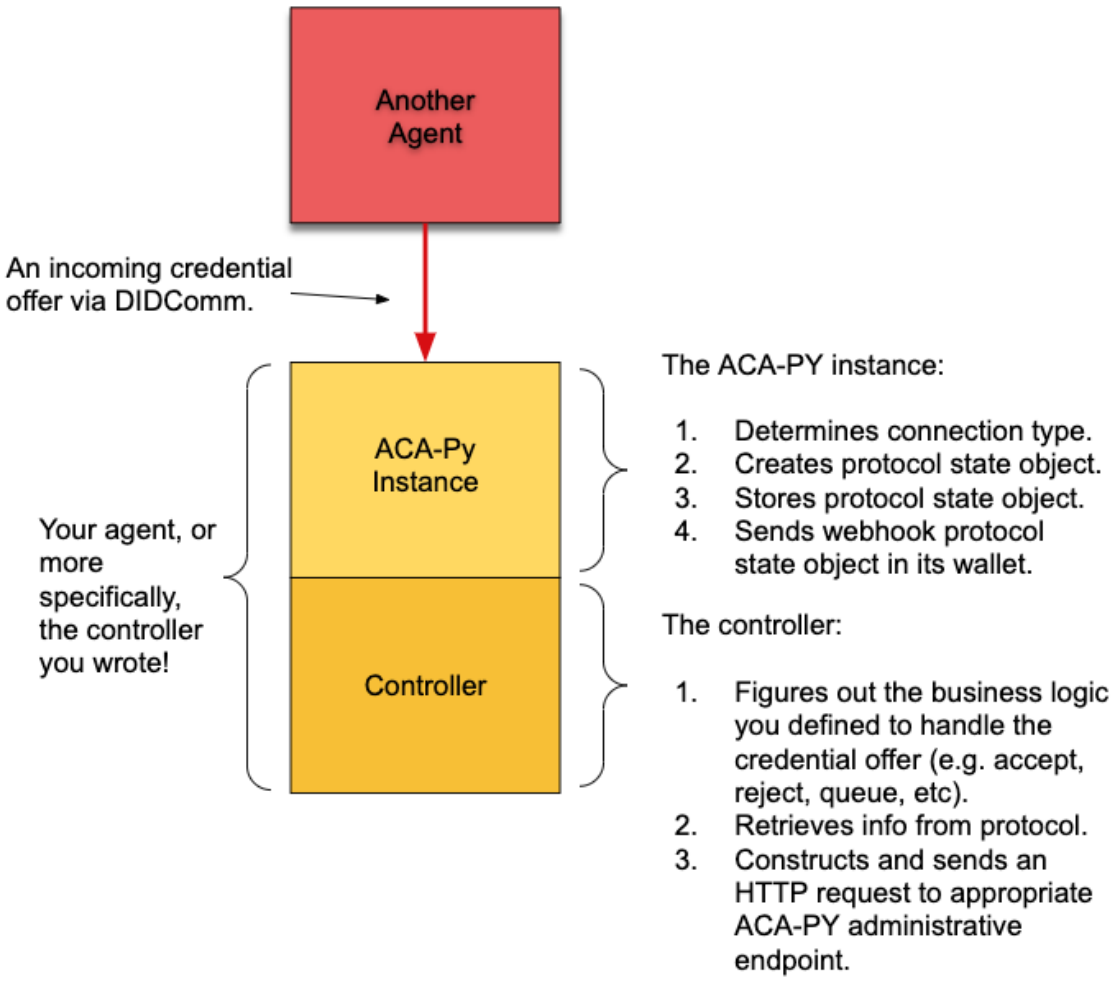
We've defined all of the pieces of a protocol, and where the code lives in an Aries framework such as ACA-Py. Let's take a look at a protocol from the controller's perspective.

The agent has started up (an ACA-Py instance and its controller), and everything has been initialized. Some connections have been established but nothing much is going on. The controller is bored. Suddenly, a DIDComm message is received from another agent. It's a credential offer message! We have to start a new instance of the "issue credential" protocol (RFC 0036). Here's what happens:

- The ACA-Py instance determines what connection is involved and creates a protocol state object. It sets some data in the object and sends a webhook (HTTP request) to the controller with information about the message and the protocol state object.
  – This assumes that the ACA-Py instance is NOT set up to "**–auto-respond-**

credential-offer." In that case, the ACA-Py instance would just handle the whole thing and the controller would just get the webhook, but not have to do anything. Sigh…back to being bored.
– Since the ACA-Py instance doesn't know how long the controller will take to tell it what to do next, the ACA-Py instance saves the protocol state object in its wallet and moves on to do other things—like waiting for more agent messages.

- The controller panics (OK, it doesn't…). The controller code (that you wrote!) figures out what the rules are for handling credential offers. Perhaps it just accepts (or rejects) them. Perhaps it puts the offer into a queue for a legacy app to process and tell it what to do. Perhaps it opens up a GUI and waits for a person to tell it what to do.
  – Depending on how long it will take to decide on the answer, the controller might persist the state of the in-flight protocol in a database.
- When the controller decides (or is told) what to do with the credential offer, it retrieves (if necessary) the information about the in-flight protocol and constructs an HTTP request to send the answer to the appropriate ACA-Py administrative endpoint. In this example, the controller might use the "**credential_request**" endpoint to request the offered credential.



Aries Protocols in ACA-Py from the Controller's Perspective

- Once it has sent the request to the ACA-Py instance, the controller might persist the data about the interaction and then lounge around for awhile, if it has nothing left to do.
- The ACA-Py instance receives the request from the controller and gets busy. It retrieves the corresponding protocol state object from its wallet and constructs and sends a properly crafted “credential request” message in a DIDComm message to the other agent, the one that sent the credential offer.
  – The ACA-Py agent then saves the protocol state object in its wallet again as it has no idea how long it will take for the other agent to respond. Then it returns to waiting for more stuff to do.

That describes what happens when a protocol is triggered by a message from another agent. A controller might also decide it needs to initiate a protocol. In that case, the protocol starts with the controller sending an initial request to the appropriate endpoint of the ACA-Py instance’s HTTP API. In either case, the behavior of the controller is the same:

- Get notified of an event either by the ACA-Py instance (via a webhook) or perhaps by some non-Aries event from, for example, a legacy enterprise app.
- Figure out what to do with the event, possibly be asking some other service (or person).
  – This might put the controller back to a waiting for an event state.
- Send a response to the event to the ACA-Py instance via the administrative API.

In between those steps, the controller may need to save off the state of the transaction.

As we’ve discussed previously, developers building Aries agents for particular use cases focus on building controllers. As such, the event loop above describes the work of a controller and the code to be developed. If you have ever done any web development, that event loop is going to look very familiar! It’s exactly the same.

Aries agent controller developers must understand the protocols that they are going to use, including the events the controller will receive, and the protocol’s administrative messages exposed via the HTTP API. Then, they will write controllers that loop waiting for events, deciding what to do with the event, and responding.

And that’s it! We’ve covered the basics of protocols from the perspective of the controller. Let’s see controllers in action!

Lab: Alice Gets a Credential

In this section, we’ll start up two command line agents, much as we did in Chapter 2. However, this time, one of the participants, Faber College, will carry out the steps to become an issuer (including creating objects on the ledger), and issue a credential to the other participant, Alice. As a holder/prover, Alice must also connect to the ledger. Later, Faber will request a proof from Alice for claims from the credential, and Alice will oblige.

We'll do this a couple more versions of this interaction in subsequent labs. There is not much content to see in this chapter of the course, but there is lots in the labs themselves. **Don't skip them!**

Click here to access the lab.

Links to Code

There are a number of important things to highlight in the code from the previous lab—and in the ones to come. Make sure when you did the previous lab, that you followed the links to the key parts of the controller code. For example, in the previous lab, Alice and Faber ACA-Py agents were started, and it's helpful to know for each what ACA-Py command line parameters were used. Several of the labs that will follow include a comparable list of links that you can use to inspect and understand the code.

Learning the ACA-Py Controller API using OpenAPI (aka Swagger)

Now that you have seen some examples of a running controller, let's get minimal. As noted earlier, ACA-Py has been implemented to expose an HTTP interface to the controller—the "Admin" interface. To make it easy for you to understand that HTTP interface, ACA-Py automatically generates an industry standard OpenAPI (also called Swagger) configuration. In turn, that provides a web page that you can use to see all the calls exposed by a running instance of ACA-Py, with examples and the ability to "try it"—that is, execute the available HTTP endpoints. Having an OpenAPI/Swagger definition for ACA-Py also lets you do cool things such as generate code (in your favorite language) to create a skeleton controller without any coding. If you are new to OpenAPI/Swagger, here's a link to what it is and how you can use it. The most important use? Being able to quickly test something out in seconds just by spinning up an agent and using OpenAPI/Swagger.

With ACA-Py, the exposed API is dynamic for the running instance. If you start an instance of ACA-Py with one or more external Python modules loaded (using the "**–plugin <module>**" command line parameter), those modules must add administrative endpoints to the OpenAPI/Swagger definition so that they are visible in the OpenAPI/Swagger interface.

Lab: Using ACA-Py's OpenAPI/Swagger Interface

In this lab, we'll use the OpenAPI/Swagger interface to interact with ACA-Py instances so you can really start to understand how to write your own controller for ACA-Py that handles your specific use case. The only controller is the OpenAPI/Swagger user interface, and you will manually invoke the API calls in sequence (using the "try it" link) to go through the same Faber and Alice use case. It's up to you to make sure Alice gets her credential!

Click here to run the OpenAPI/Swagger lab.

Lab: Help Alice Get a Job

Time to do a little development. The next assignment is to extend the command line lab with Alice and Faber to include ACME Corporation. Alice wants to apply for a job at ACME. As part of the application process, Alice needs to prove that she has a degree. Unfortunately, the person writing the ACME agent's controller quit just after getting started building it. Your job is to finish building the controller, deploy the agent and then walk through the steps that Alice needs to do to prove she's qualified to work at ACME.

Alice needs your help. Are you up for it? Click here to run the lab.

Lab: Python Not For You?

The last lab in this chapter provides examples of controllers written in other languages. GitHub user amanji (Akiff Manji) has taken the agents that are deployed in the command line version of the demo and written a graphical user interface (GUI) controller for each participant using a different language/tech stack. Specifically:

- Alice's controller is written in Angular.
- Faber's controller is written in C#/.NET.
- ACME's controller is written in NodeJS/Express.

To run the lab, go to the instructions here.

As an aside, Akiff decided to make these controllers based on an issue posted in the ACA-Py repo. The issue was labelled "Help Wanted" and "Good First Issue." If you are looking to contribute to some of the Aries projects, look in the repos for those labels on open issues. Contributions are always welcome!

Building Your Own Controller

Want to go further? This is optional, but we recommend doing this as an exercise to solidify your knowledge. Build your own "Alice" controller in the language of your choice. Use the pattern found in the two other Alice controllers (command line Python and Angular) and write your own. You might start by generating a skeleton using the OpenAPI/Swagger tools, or just by building the app from scratch. No link to a lab or an answer for this one. It's all up to you!

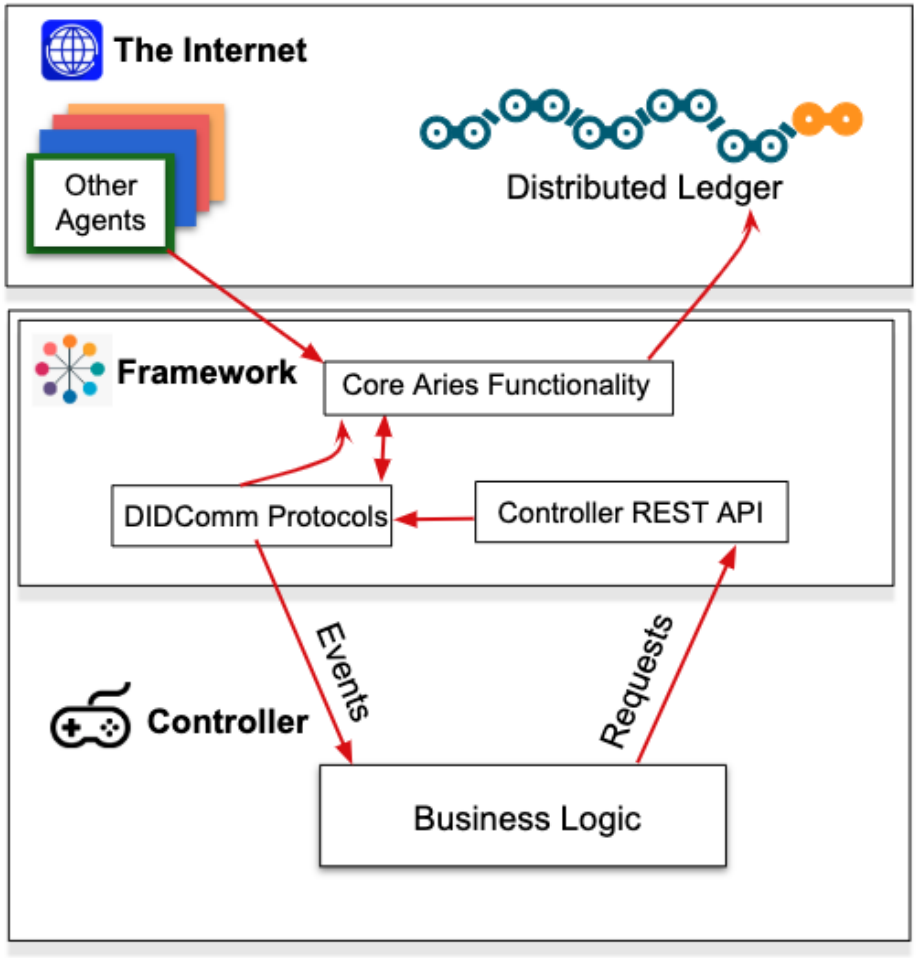
If you build something cool, let us know by clicking here and submitting an issue. If you want, we might be able to help you package up your work and create a pull request (PR) to the **aries-acapy-controllers** repo.

Controllers for Other Frameworks

In this chapter we have focused on understanding controllers for **aries-cloudagent-python** (ACA-Py). The Aries project has several frameworks other than ACA-Py in various stages of development. In this section, we'll briefly cover how controllers work with those frameworks.

aries-framework-dotnet

The most mature of the other Aries frameworks is **aries-framework-dotnet**, written in Microsoft's open source C# language on the .NET development platform. The architecture for the controller and framework with **aries-framework-dotnet** is a little different from ACA-Py. Instead of embedding the framework in a web server, the framework is built into a library, and the library is embedded in an application that you create. That application is equivalent to an ACA-Py controller. As such, the equivalent to the HTTP API in ACA-Py is the API between the application and the embedded framework. This architecture is illustrated below.



Aries Agent Architecture (**aries-framework-dotnet**)
Licensed under CC BY 4.0

As we'll see in the chapter on mobile agents, **aries-framework-dotnet** can be embedded in Xamarin and used as the basis of a mobile agent. It can also be used as a full enterprise agent, or as a more limited cloud routing agent. A jack-of-all-trades option!

aries-framework-go

The **aries-framework-go** takes the same approach to the controller/framework architecture as ACA-Py—an HTTP interface between the two. In fact, as the team building the framework has created the implementation, they have used the same administrative

interface calls as ACA-Py. As such, a controller written for ACA-Py should work with an instance of **aries-framework-go**.

What's unique about this framework is that it is written entirely in golang without any non-golang dependencies. That means that the framework can take advantage of the rich golang ecosystem and distribution mechanisms. That also means the framework does not embed the **indy-sdk** (libindy) and as such does not support connections to Indy ledgers or the Indy verifiable credentials exchange model. Instead, the team is building support for other ledgers and other verifiable credential models. It's likely that support could (and may in the future) be added for Indy, but that might be at the cost of the "pure golang" benefits inherent in the current implementation.

Other Open Source Aries Frameworks

As we are creating this course, other Aries open source frameworks or Aries SDKs are being written in other languages and tech stacks. These include Ruby, Java and JavaScript. Check on the Aries Work Group wiki site to find what teams are building those capabilities, how they have structured their implementations, and how you can start building a controller on top of their work. Or how you can help them build out the frameworks.

Summary

We've covered an awful lot in this chapter! As we said at the beginning of the chapter, this is the core of the course, the hands-on part that will really get you understanding Aries development and how you can use it in your use cases. The rest of the course will be a lot lighter on labs, but we'll refer back to the labs from this chapter to put those capabilities into a context with which you have experience.

Chapter 5. Digging Deeper—The Aries Protocols

Chapter Overview

In the last chapter we focused on how a controller injects business logic to control the agent and make it carry out its intended use cases. In this chapter we'll focus on what the controller is really controlling—the messages being exchanged by agents. We'll look at the format of the messages and the range of messages that have been defined by the Aries Working Group. Later in the course, when we're talking about mobile agents, we'll look deeper into how messages get from one agent to another.

Some of the topics in this chapter come under the heading of "DIDComm Protocol," where pairs (and perhaps in the future, groups) of agents connect and securely exchange messages on DIDs each has created and (usually) shared privately. The initial draft and implementations of the DIDComm protocol described here were incubated in the Hyperledger Indy Agent Working Group, and later the Aries Working Group. As this course is being written, some of the maintenance and evolution of DIDComm is being transferred from Aries to the DIDComm Working Group within the Decentralized Identity

Foundation (DIF). This transfer is being done because Hyperledger is an open source code creating organization, not a standards creating body. Part of DIF's charter is to define standards, making it a more suitable steward for this work. Of course, open source implementations of the standards will remain an important part of the Hyperledger Aries project.

Learning Objectives

As we've just explained, this chapter is all about messaging and the protocols that allow messaging to happen. We will discuss:

- The **aries-rfcs** repository—the home of all the documents!
- The two layers of Aries messaging protocols—or "DIDComm 101."
- Which protocol layer a developer needs to worry about (hint: it's the Aries protocols layer).
- The format of protocol messages.
- Message processing in general.
- The Aries Interop Profile (AIP)—to manage versions of the different protocols for agent interoperability.

The All-Important aries-rfcs Repository

The concepts and features that make up the Aries project are documented in the Hyperledger **aries-rfcs** GitHub repository. RFC stands for "request for comments" and is a type of text document used by a wide range of technical groups to create understanding towards defining and evolving communication standards. RFCs are the documents that tell developers how they MAY, MUST and SHOULD specify things to meet a particular standard. With Aries, the RFCs are defined into the two groups of concepts (background information that crosses all protocols) and features (specifications of specific protocols).

The **aries-rfcs** repository is full of extremely detailed information about exactly how each protocol is to be implemented. While the repo is a crucial resource for developers, it is overwhelming for newcomers. It's definitely not the best place to get started—kind of like learning a new language by reading the dictionary. It's important that you are aware of the **aries-rfcs** repo so that when you are deep in the weeds and need to know exactly what a certain protocol does, you know where to look. For now though, follow along here and we'll get you going!

Through the remainder of the course, we'll provide pointers into the **aries-rfcs** repo for things we mention. That way, when you are on your own, you'll know how to navigate the repo when you need to know the crucial details it holds. Here's a first example: the index markdown file in the root of the repo provides a generated list of all of the RFCs in the repo ordered by their current maturity status, proposed, demonstrated, accepted, etc. Check it out!

Note: Hyperledger Indy has a comparable repository to **aries-rfcs** called **indy-hipe** (for Hyperledger Indy Project Enhancement). Despite the name difference, the purpose of the

repository is the same. Further, as some of the DIDComm standards move to the Decentralized Identity Foundation (DIF), some of the documents will move there as well. Rest assured, as things are moved from **aries-rfcs** to DIF, pointers to the latest information will be available in **aries-rfcs**.

Basic Concepts of DIDComm Messaging

We introduced the core concepts of DIDComm in Chapter 5 of *LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa*, a prerequisite of this course. Here is a review of the key points covered there to provide context as we move on to the developer's details of Aries messaging.

So, Back to Naming…

We hate to do this, but we need to start this section by talking about naming. Again.

As noted before, we introduced the core concepts of DIDComm in Chapter 5 of the prerequisite course to this one (LFS172x). In that, we used the terms "envelope protocol" (getting messages from a sender to a receiver agent) and "content protocols" (the content of the messages to accomplish a task). Since then, the community has moved on to use other terms. The concepts are the same, but in this course, we're going to use some different names. Instead of envelope protocol, we'll use "DIDComm protocol," and instead of "content protocols," we're going to use "Aries protocols." We can't promise the terms won't change again. But the concepts will remain solid:

- There's a way to securely send messages between agents.
- There are specific sequences of messages (protocols) that allow agents to collaborate on a shared task (such as issuing a credential or proving claims).

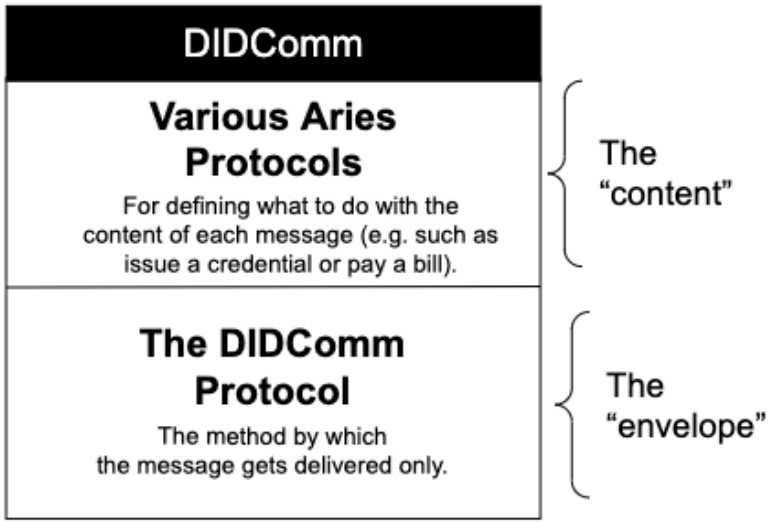
As Ferris Bueller famously said, "Life moves pretty fast." This is true in Aries.

DIDComm Concepts

The core capability of an Aries agent is peer-to-peer messaging—connecting with agents and sending and receiving messages to and from those agents to accomplish some interaction. The interaction might be as simple as sending a text message, or more complex, such as negotiating the issuing of a verifiable credential or the presentation of a proof.

Enabling both the exchange and use of messaging to accomplish higher level transactions requires participants interact in pre-defined ways, to interact by following mutually agreed upon protocols. Aries protocols are just like human protocols, a sequence of events that are known by the participants to accomplish some shared outcome. For example, going out to a restaurant for a meal is a protocol, with both the guests and the restaurant staff knowing the sequence of steps for their roles—greeting, being seated, looking at menus, ordering food, etc. Unlike human protocols (etiquette), Aries protocols need to be carefully specified and then implemented by multiple participants.

With Aries, there are two levels of messaging protocols. Since all of the messaging is based on the exchange and use of DIDs, the messaging mechanism is called **DID Communication** or **DIDComm** for short.



DIDComm
Licensed under CC BY 4.0

At the lower level is the **DIDComm protocol**, the method by which messages are exchanged, irrespective of the message content. You can think of the DIDComm protocol as being like the postal service. You write a letter, place it into an envelope, address it to a recipient, give it to the postal service and magically, it appears at the recipient's door. And, just like the postal service, there is a lot of complexity in the sending and receiving of a message.

At a higher level are the **Aries protocols**, hundreds of protocols that define back-and-forth sequences of specific messages to accomplish some shared goal. The simplest of these is one agent sending another agent a text message ("Hi, how are you?"). A much more complex protocol is issuing an Indy-style credential, where it takes at least three messages back and forth to offer, request and issue a credential. The set of messages, the roles of the participants, plus some extra details such as acknowledgments and error handling, define each of the many Aries protocols.

With the DIDComm protocol we don't care about the content of the message, just the method by which the message gets delivered. With the Aries protocols it's the reverse—we

know that the messages get delivered (we don't care how), and we only care about what to do with the content of each message.

Note: In the previous section we talked about the transfer of the standardization work for DIDComm Messaging from Hyperledger Aries to DIF. More specifically, it is the DIDComm protocol standardization work that is moving. It's possible a couple of what we are calling the Aries protocols may move as well—notably the protocols around establishing and maintaining a secure DID-based communication channel.

What Protocols Matter for Development?

As we've discussed, there are two levels of protocols at play with DIDComm messaging: the DIDComm protocol (aka "envelope") and the Aries protocols (aka "content"). While it's important to understand that both layers of protocols exist, for the most part, Aries application developers (those building Aries controllers) are really not exposed to the lower DIDComm (envelope) layer at all. The DIDComm (envelope) layer is handled almost entirely in code provided by the Aries frameworks.

On the other hand, Aries application developers do need to understand a lot about the Aries protocols. While the mechanics of receiving Aries protocol messages and preparing and sending messages is handled by the code in the Aries frameworks, the semantics (business processing) of what to do with those messages is up to the controller. For example, when an enterprise agent wants to issue a credential to another agent, the controller doesn't have to get into the nitty-gritty of preparing a verifiable credential. However, the controller does have to pass in (via the controller-to-framework API) the claims data that will be put into the credential.

As such, for the remainder of this chapter, we will focus only on the upper-level, Aries protocols. We will cover a bit about the lower level DIDComm (envelope) protocol in Chapter 7 when we talk about message routing in relation to mobile agents.

The Format of Aries Protocol Messages

As we saw in the labs in the previous chapter, the format of Aries protocol messages is JSON. Let's start our look into Aries protocols by considering an example of a simple message and going through each of the JSON items. The following is the only message in the "basic message" protocol defined in Aries RFC 0095.

```
{
   "@id": "123456780",
   "@type":
"did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/basicmessage/1.0/message",
   "~l10n": { "locale": "en" },
   "sent_time": "2019-01-15 18:42:01Z",
   "content": "Your hovercraft is full of eels."
}
```

The purpose of the message is simple—to send a text message from one agent to another.

While the format is standard JSON, some conventions are used in messages such that they are compatible with JSON-LD (JSON Linked Data)—notably the "**@type**" and "**@id**" items that are in every Aries protocol message. For those unfamiliar, JSON-LD is based on JSON and enables semantic information (what the JSON items mean) to be associated with JSON. It is a W3C-standard way to embed in the JSON itself a link to metadata about the items in the JSON structure so that an application developer can know exactly how the data is to be processed. "Compatible with JSON-LD" means that while the JSON does not support all the features of JSON-LD, it respects JSON-LD requirements such that JSON-LD tools can process the messages. It's not crucial to know more about Aries messaging and JSON-LD at this point, but if you are curious, you can read Aries RFC 0035 about support for JSON-LD in Aries messages.

Back to the message content. Every message in every protocol includes the first two fields:

- **@id** is a unique identifier for the message, often generated from a globally unique ID (GUID) library.
- **@type** is the type of the message. It is made up of the following parts:
  – **Namespace** (e.g. **did:sov:BzCbsNYhMrjHiqZDTUASHg;spec**). See details below.
  – **Protocol** (e.g. **basicmessage**). The name of protocol to which the message type belongs.
  – Message **version** (e.g. **1.0**). Aries uses the semver versioning system, although with just major.minor version numbers components.
  – Message **type** (e.g. **message**). The actual text of the message.

Namespace is defined so that when appropriate, different groups (e.g. companies, standards groups, etc.) can independently define protocols without conflicting with one another. All of the protocols defined by the Aries Working Group (and hence, documented in the **aries-rfcs** repo) use the same namespace, the one listed above (**did:sov:BzC**…). The plan when that specific DID was created was that by resolving the DID and getting the "spec" service endpoint from the DIDDoc, the **@type** value could be transformed into an HTTP link pointing to the specification of the message protocol. However, that DID was never registered on a public ledger, and so it cannot be resolved. In December 2019, the Aries Working Group decided to change **namespace** from being a DID to being an HTTP link. For the Aries Working Group, the **namespace** URL is planned to be http://didcomm.org. As such, some time in 2020 you will see that HTTP address replace the DID listed above. At the time of this writing, it's not clear when the HTTP URL will actually link to the message protocol specification as found in the **aries-rfcs** repository.

The remaining items in the JSON example above are specific to the type of the message. In our example, these are the items **~l10n**, which we'll talk about in the next section, **sent_time** (when the message was sent, as defined by the sender) and **content** (the text of the message). For each message type in each protocol, the protocol specification defines all of the relevant information about the message type specific item. This includes which items are required and which are optional, the data

format (such as the time format in the **sent_time** field above), and any other details necessary for a developer to implement the message type. That should be pretty clear, but, since it's a good idea to get used to going to the **aries-rfcs** repository to get all the details on topic, check out RFC 0020 to see the accepted information about message types.

Message Decorators

In addition to protocol specific data elements in messages, messages can include "decorators," standardized message elements that define cross-cutting behavior. "Cross-cutting" means the behavior is the same when used in different protocols. Decorator items in messages are indicated by a "**~**" prefix. The **~l10n** item in the previous section is an example of a decorator. It is used to provide information to enable the localization of the message—translations to other languages. Decorators all have their own RFCs. In this case, the localization decorator is specified in RFC 0043.

The most commonly used decorator is the "**~thread**" (RFC 0008) decorator, which is used to link the messages in a protocol instance. As messages go back and forth between agents to complete an instance of a protocol (e.g. issuing a credential), **~thread** decorator data lets the agents know to which protocol instance the message belongs, and the ordering of the message in the overall flow. It's the **~thread** decorator that will let Faber College issue thousands of credentials in parallel without losing track of which student is supposed to get which verifiable credential. We'll talk later in this chapter about the **~thread** decorator when we go over the processing of a message by an agent. You'll note that the example message in the previous section doesn't have a **~thread** decorator. That's because it is the first message of that protocol instance. When a recipient agent responds to a message, it takes the "**@id**" of the first message in the protocol instance and makes it the **@thid** (thread-id) of the **~thread** decorator. Want to know more? As always, the place to look is in the **aries-rfcs**. Other currently defined examples of decorators include **~attachments** (RFC 0017), **~tracing** (RFC 0034) and **~timing** (RFC 0032).

Decorators are often processed by the core of the agent, but some are processed by the protocol message handlers. For example, the **~thread** decorator is processed by the core of the agent to connect an incoming message with the state of a protocol that is mid-flight. On the other hand, the **~l10n** decorator (enabling text display in multiple languages) is more likely to be handled by a specific message type.

Special Message Types: Ack and Problem Report

The Aries Working Group has defined two message types that deserve special attention: **ack** (acknowledge) and **problem_report**. Much like decorators, these are cross-cutting messages, used in the same way across many protocols. In particular, these messages are for error handling.

In executing an instance of a protocol, there are often times when one agent wants to say to the other "yup, I got your last message and all is well." Likewise, if something goes wrong in a protocol, an agent will want to notify the other "uh-oh, something bad has happened!" Rather than having each protocol define their own special purpose messages,

the common **ack** and **problem_report** messages were defined so that any protocol can adopt those messages. When a protocol adopts these messages it means that there is a message type of **ack** (and/or **problem_report**) in the protocol, even though there is not a formal definition of the message type in the specification. The definition is found in the the respective RFCs: RFC 0015 for ack, and RFC 0035 for problem report.

Remember that unlike HTTP requests, all DIDComm messages are asynchronous. An agent sends off the message, doesn't immediately get a response, and continues doing other things until it receives the next message of the protocol. Thus, a common use case for **problem_report** is when an agent sends off a message and doesn't hear back in a time it thinks is reasonable, it can send a problem report message asking if the previous message was received.

Framework Message Processing

Messages are sent between agents. Let's talk about the processing of messages from the perspective of the controller. We'll start with a controller initiating a message, and then move onto the receipt of a response to the message.

You should have noticed in looking at the ACA-Py startup parameters and in the labs in the last chapter that ACA-Py can handle a number of common interactions without involving a controller. For example, when ACA-Py is started with the **–auto-accept-invites** parameter, ACA-Py will not wait on the controller to tell it what to do and will just accept the invitation. Those parameters are to simplify getting started and debugging ACA-Py itself. It is rare that those parameters would be used in a production agent implementation. **Don't rely on those parameters in your application!** In this section we'll make sure those options are off.

Initiating a Protocol

We'll start with a protocol initiated by the controller. Assume that Faber College and Alice have established a DIDComm connection and now Faber College wants to offer Alice a verifiable credential about her accomplishments at Faber. The Faber controller uses the

administration API to initiate the process by requesting the agent framework send a message identifying the:

- type of message to be sent
- content of the message
- connection to which the message is to be sent

In ACA-Py (and likely other Aries frameworks) the type of message to be sent is implied by the API endpoint used by the controller. In general, there is one API endpoint per message type. The content is provided as JSON data as defined by the API endpoint. Usually, the JSON provided by the controller corresponds to the content items of the message that is to be sent—all of the JSON message fields except **@id** and **@type**. In addition, a connection ID is sent that the agent framework code can use to find the details of the connection—an identifier for another agent.

Note: In order to know the connection on which to initiate a protocol, the controller must keep a record of the connections as they are established, perhaps storing the relationship as an attribute of an account it holds. For example, Faber College might keep correspondence between Alice's record and the IDs of the DIDComm connections it has established with students in its "Student Information System."

Armed with that information, agent framework code carries out the details of:

- Finding the connection for Alice.
- Routing messages to Alice's agent.
- Generating an **@id** for the message.
- Packaging (including encrypting) the message.
- Sending the credential offer message.

If the message requires any special processing, such as getting information from a public ledger, or calling the **indy-sdk**, the agent framework code handles that as well. Since usually a first message is not going to be the only message of the protocol (e.g. Alice's agent is expected to reply to the message), the agent framework also persists a protocol state object in anticipation of the response to come. We'll talk more about that in the next section.

Receiving a Response

After the message is sent, the agent handles other work it needs to do. For an enterprise agent such as Faber's, that might be hundreds or hundreds of thousands of messages being exchanged with the agents of other users. The message from Alice might take a long time to come back. For example, the message might go to Alice's mobile agent while her phone is not connected to the Internet because she is climbing a mountain in the Andes. Alice won't receive the message until her phone is back online, and even then, she won't respond until she's had a long sleep to recover from her amazing mountain adventure. Plus she needs to catch up on Instagram…

Eventually, a response from Alice's agent is received—a credential request message because Alice wants to be issued a credential about her degree from Faber College. The agent framework code processes the message (with help from the controller) as follows:

- It uses the **~thread** information to find the message's corresponding protocol state object and retrieves it from storage. In our example, the state object from the initial credential offer message.
  – If it doesn't find the protocol state, there is an error in the message metadata, or if the message is found to be out of order, it will likely send an event to the controller to determine if it should send a problem report message back to Alice's agent.
- It processes any generic decorators, ones not intended to be processed by the message type handler.
- It hands the message and the protocol state object to the appropriate message type handler. In this case, the credential request handler that is part of the issue credential protocol is invoked.
- The message type handler processes the message, updates the protocol state object appropriately and sends an HTTP request to the controller (a webhook) with the protocol state object.
- The controller receives the information via the event webhook.
- Since the agent framework code doesn't know how long it will take for the controller to respond to the event, it again persists the protocol state object and carries on with its other duties.
  – Even some enterprise controllers might take a long time to respond. For example, perhaps Faber College Student Information System is not connected to their Accounting System. Before they issue a credential, they have to email Accounting to see if Alice is fully paid up to the college and wait on a response. That could take hours…
- The controller uses the information from the protocol state to figure out how it wants to respond to the message. In this case, it checks Alice's data in the Student Information System and decides if it's going to issue the credential and the claims that are to go in the credential.
- The controller issues another call to the Aries administration API with information on how to respond to the previous message. It passes all the same types of information as with the first message (above), with the content including the claims to be put into Alice's verifiable credential. It also passes a reference to the protocol state object.

- The agent framework code receives the request, retrieves the protocol state object, prepares and sends the message to Alice and updates and saves the protocol state object.

You're right—there's a lot going on there! The good news is that sequence described above is the same for almost every request/response transaction, and it is very much like the processing of traditional web servers—one layer handles the mechanics of receiving requests and responding, and another layer figures out what the request is for and how to respond.

Lab: Executing a Protocol

We only have one lab in this chapter, but it's a doozy! We're going to use the OpenAPI/Swagger interface for ACA-Py that was introduced in the last chapter to manually walk through a connection and credential issuance process from Faber to Alice. There's no controller other than you and OpenAPI/Swagger to receive and respond for each of the agents.

Click here to jump into the lab.

The Aries Interop Profile (AIP)

As you can see from looking at the list of RFCs in the **aries-rfcs** repo, there are a lot of detailed specifications in a lot of documents. Over time, changes to the specifications will result in different versions of the documents, varying on both status (Proposed, Demonstrated, etc.) and protocol version numbers. That evolution is a good thing—it means the technical details are improving over time as contributors learn more and more.

However, there is a bad side to all that change as well. If there are many implementers building agent frameworks independently, how will they know what versions of the different protocols to use to be interoperable with the rest of the community? This is a common problem in many technical communities building protocols—Wifi makers, Bluetooth, OAuth and more have all faced this challenge. The Aries community needs a way for implementers to target the same versions of the protocols to enable interoperability. As well, the community needs a way to evolve that target to use new and updated protocols as they stabilize. The mechanism the Aries community has chosen to do that is the Aries Interop Profile (AIP). AIP is much like the various published WiFi standards (e.g 802.11a, 802.11b/g/n and 802.11ac and so on).

An AIP is a versioned (e.g.1.0) profile that defines a list of supported RFCs and links to the precise version (the literal github commit) of each RFC. Using this mechanism, the protocols themselves can continue to evolve as the community contributes to them. At the same time, the community of agent framework builders can build solutions to support the protocols in an AIP version and expect that agents built on their framework will work with agents that support the same AIP. Further, when the community of framework builders decide to support a new set of protocols and protocol versions, they can define a new AIP version and a migration path to that new version.

How does this affect a developer building agent controllers?

- First, it reduces the number of protocols that you need to learn. As of writing this course, the current AIP version, AIP 1.0, includes only 19 RFCs and only six are pure Aries messaging protocols. That means that despite the many RFCs in the **aries-rfcs** repo, the subset of "getting started" RFCs is quite small.
- Second, as you select an agent framework to use, you should be aware of the framework's support for the current AIP version and plan for supporting any defined future AIP versions.
- Finally, while most of the hard work in supporting an AIP version is in the agent framework, controllers will be impacted in the form of changes to the administrative API. Thus, as a controller developer you should be aware of when AIP version changes are coming and their potential impact on your controllers. That means that you need to be plugged into the Aries Working Group, as that is the group that defines when new versions of AIPs will be introduced.

As you might expect, there is an RFC for that! Here's the link to the AIP RFC 0302. Embedded in the document are the current AIP versions, and (possible) proposals for the next AIP versions.

Summary

We have covered a lot of information about the Aries protocols including the details of the protocols you will use as you code them into your controller. As you build your own application and need to know more about the underlying details of the protocols, you know where to look, right—the **aries-rfcs** repo!

Let's highlight the key takeaways from this chapter:

- You need to know about the **aries-rfcs** repos but you don't need to do a deep dive (or even a shallow one) there until you have something specific to look up (e.g. handling a particular protocol).**Note**: Want to gain some points in the Aries community? If you find something in an RFC that needs clarification, submit an issue or a pull request to make things better. Such contributions are always welcome.
- With Aries, there are two levels of messaging protocols. Since all of the messaging is based on the exchange and use of DIDs, the overall messaging mechanism is called **DIDComm** (for DID communication).
  – The DIDComm protocol handles the delivery of messages.
  – The Aries protocols define the content of the messages delivered.
- As an Aries application developer your focus is on the Aries protocols, the protocols that define back-and-forth sequences of specific messages to accomplish some shared goal. You don't need to know much about how the messages get delivered, just about the content of each message.
- As a developer, you need to be aware of the Aries Interop Profile (AIP) to keep track of the versions of the Aries protocols that other developers and

organizations are using so that your applications will be able to interoperate with theirs.

Easy, peasy, right?! You're well on your way to becoming an Aries developer!

Chapter 6. Aries Development Tools

Chapter Overview

We'll take a break in this chapter from diving into the weeds of controllers, APIs and protocols and talk about a few development tools available in the Aries community. At the time of writing, these tools are relatively new and contributions are very much welcome. This is a short chapter but it might spark some ideas about how to extend these tools, or what other tools might be helpful.

Learning Objectives

In this chapter, we will describe the current suite and state of the various Aries development tools. You will learn about:

- The Aries Toolbox
- The Aries Protocol Test Suite
- The Aries Agent Test Harness

We also have some labs that look at what's available today on using these tools.

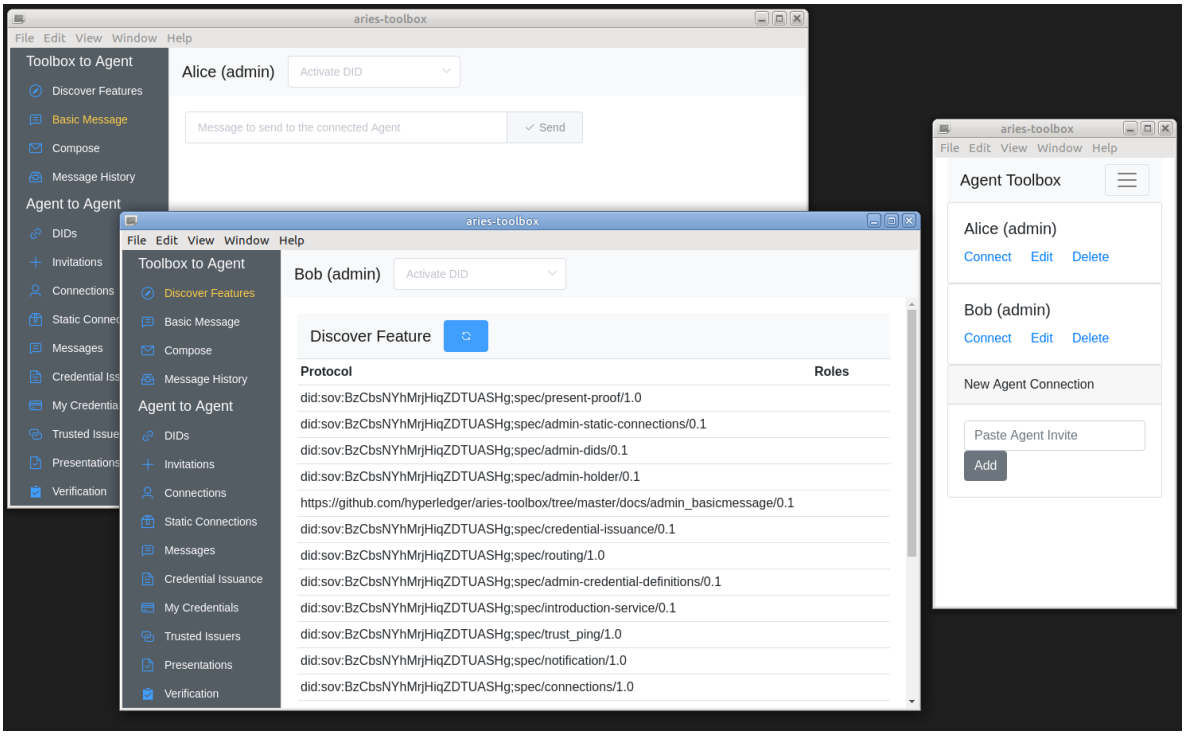
The Aries Toolbox

The Aries Toolbox is a desktop tool (written in Electron using Vue) that allows a user to control the behavior of running Aries agents. It provides a graphical user interface tuned to Aries for developers and system administrators to control (in theory) any running Aries agent. To enable an instance of the Aries Toolbox to control running agents, additional protocol message types are added to these agents. With that additional functionality, an ACA-Py agent can be controlled by an Aries Toolbox instance using the DIDComm channel, instead of by the HTTP interface used by a typical ACA-Py controller.

Conceptually, the Aries Toolbox enables controller functionality similar to what we saw with ACA-Py and its OpenAPI interface. However, unlike the generic OpenAPI interface, which

is used with any HTTP interface, the Aries Toolbox has Aries-specific knowledge and capabilities, making it easier to accomplish certain agent-related functions. As such, while the OpenAPI interface we talked about in earlier chapters is strictly for developers to learn about the administrative API to ACA-Py, the Aries Toolbox can be used to connect to any running agent—even those in production—to accomplish administrative tasks.

During interactions with other agents, the Aries Toolbox shows connection, protocol state and debugging information. The image below is the Aries Toolbox running with connections to ACA-Py agents for Alice and Bob.



Aries Toolbox Screenshot

As shown in the image, an administrative interface for each agent connection is presented in its own window, along with a list of actions that the user can trigger the agent to perform. The list of actions is tuned to each connected agent by the Aries Toolbox executing the "Discover Features" protocol (RFC 0031).

Architecturally, Aries Toolbox works quite differently from the combined ACA-Py/OpenAPI administrative interface we looked at earlier in the course. Rather than using the HTTP interface that ACA-Py exposes, Aries Toolbox uses the same DIDComm agent interface supported by every Aries agent. To work with a given agent, Aries Toolbox requires that the connected agent support special administrative message types. It's through these added message types that the Aries Toolbox is able to control the agent. For ACA-Py, a second repo, **aries-acapy-plugin-toolbox**, provides an external Python module that can be included in any ACA-Py instance at runtime (using the **–plugin** ACA-Py command line parameter) to add the necessary admin message types. The following contrasts the ACA-Py Admin API with the approached using by the Aries Toolbox:

ACA-Py/OpenAPI	Aries Toolbox
Uses an HTTP interface to ACA-Py.	Uses the DIDComm agent interface.

Use when you are learning about the HTTP interface to ACA-Py in order to write your own controller.	Can (in theory) connect to any Aries Agent —provided it supports extended protocols needed by Aries Toolbox.
	Use this for controlling an agent to establish connections and execute higher level protocols.

The use cases for the two techniques are quite different. The ACA-Py/OpenAPI mechanism is a training tool intended to be used by developers learning how to use the administrative API exposed by an ACA-Py agent in order to write a controller application. Aries Toolbox is for interactively controlling Aries agents. However, since it does not expose a programmable API, Aries Toolbox cannot be used as the basis of a controller application.

The Aries Toolbox implements a couple of interesting use cases:

- Users new to Aries agents can explore the interactions between agents (connections and protocols) at a higher-than-the-controller level, to understand how they work.
- Aries Toolbox can be used to perform one time agent actions that previously required the Indy Command Line Interface (CLI) or single purpose scripts.

For example, Aries Toolbox can be used for experimenting with creating and writing a new schema and credential definition to a test ledger, and then trying out issuing credentials using those objects. We'll look at more such tasks in Chapter 8 when we talk about getting ready for production.

At the time of writing this course, the Aries Toolbox is still quite new and improving quickly. The special administrative message types it uses are not part of **aries-rfcs** and are not supported by other agent frameworks. The ACA-Py capability to add Python external modules to an agent at runtime made it easy for Aries Toolbox developers to add the necessary messages to ACA-Py agent instances.

Lab: The Aries Toolbox

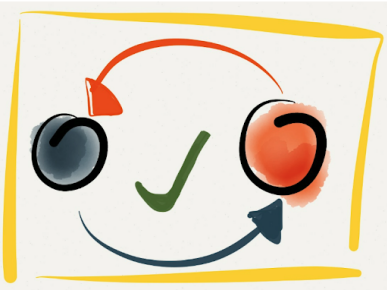
In this lab, we'll spin up some Aries ACA-Py agents and the Aries Toolbox, and use the Toolbox graphical user interface to control the agents.

Click here to run the lab.

The Aries Test Suites

In the last chapter, we talked about the Aries Interop Profile (AIP). Observant course participants might have noticed that although we mentioned that agent framework builders can claim (self-assert) support for a specific AIP version, wouldn't it be better if they could prove they support a specific AIP version? As we know from all we've learned about

verifiable credentials, self-asserted claims are not nearly as useful as claims issued by an authority!



To prove an agent or agent framework's support for a given AIP version, we want to be able to run an instance of the agent through a set of conformance tests aligned with the versions of the protocols included in the AIP. The test suite must generate, at minimum, a report on the tests executed and on conformance—or better yet, a verifiable credential with claims about the conformance results.

Unfortunately, at the time of writing this course, the Aries community has not yet produced a complete test suite that we can use (as we should be!) for AIP conformance testing. On a brighter note, two efforts have been started with the goal of being able to provide an easy tool for building and executing protocol tests. In the following, we'll look at the two approaches and run labs covering both of the test suites. It's quite possible that over time one of the two test suites will be deprecated. If that happens, we'll update this section to cover just the one suite.

Before jumping into the two test suites, we should mention that there is an RFC (0270) that outlines the requirements that should be provided by an Aries Interop Test Suite. That RFC is still (as of writing this course) in the "proposed" state. Like the two test suites, it will likely evolve as we learn more in implementing interoperability testing capabilities for Aries. The following table highlights some of the important features mentioned in the RFC that we want in a good test suite. Think about these features as you go through this section.

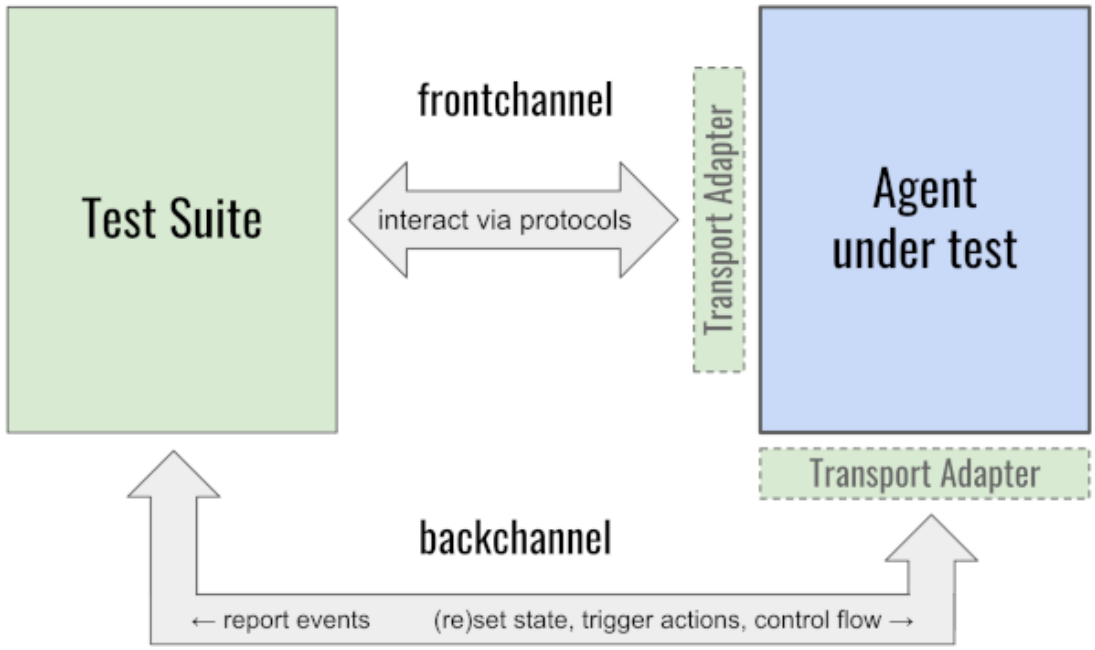
Important features for an Aries Interop Test Suite (proposed):

- Test cases should be built around protocols and the different roles supported by the protocols. For example, test cases should cover the agent both initiating and responding to a protocol.
- Tests should exercise both expected and *unexpected* inputs; agents should gracefully handle receiving bad data.
- The test suite should have test variations to exercise different features of agents.
- The test suite should allow the agent-under-test (AUT) configuration to define the set of tests to run.
- An execution of the test suite should report only on the results of the tests executed.
- The test suite should be reasonably efficient to run. This implies there is likely a need to load the state of participating agent(s) to a given starting state vs. executing actions to get to the desired starting state for every test.

- The test suite should be automated and able to be included in a CI/CD pipeline as part of the code promotion process.

A couple of these requirements (e.g., having the agent-under-test (AUT) initiate tests and loading state) imply that the test suite has a way to control the AUT for some tests—to get it to do something versus waiting to be contacted via agent messaging. This is done using the concept of a "backchannel," an API between the test suite and the AUT. This is pictured below, in an image taken from RFC 0270. The "frontchannel" is the normal DIDComm messaging path between agents, while the backchannel is used to tell the agent what to do in executing a test.

Note: In order to execute the tests in a test suite, an AUT may have to build some custom code to handle the backchannel commands sent by the test suite. This is part of deploying an instance of the agent to run the tests in the suite.



Controlling the AUT with a Backchannel
Licensed under CC BY 4.0

With that introduction into test suites, let's take a look at the two fledgling test suites in Aries today, namely the Aries Protocol Test Suite and the Aries Agent Test Harness.

Aries Protocol Test Suite

The approach taken by the Aries Protocol Test Suite (APTS) is that the test suite is itself a minimal agent that interacts with the agent-under-test (AUT) to execute test cases. The architecture is close to the picture above, including the test suite, the AUT, and using the frontchannel and backchannel to perform tests.

To run a set of tests, the AUT (for example, an instance of ACA-Py and a controller) is started, and configuration information about the AUT instance is passed to the APTS. The APTS initializes itself and uses the information from the AUT to carry out tests, typically by

establishing a DIDComm connection, invoking the protocols for the tests to be carried out, and monitoring the responses from the AUT to ensure conformance.

The repository for the Aries Protocol Test Suite (APTS) can be found here. At the time of writing, a handful of tests exist in the test suite with new ones being added regularly.

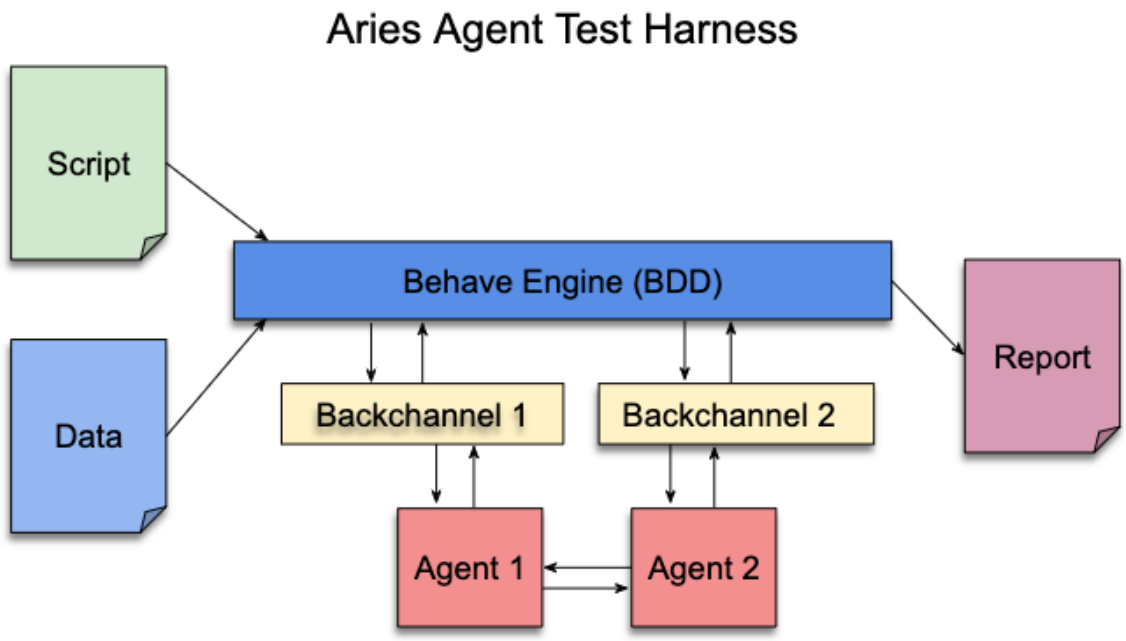
Lab: Aries Protocol Test Suite

In this lab we'll look at the APTS that is used to evaluate the conformance of an Aries ACA-Py agent to the Aries Protocols.

Click here to run the lab.

Aries Agent Test Harness

The second style of test suite being explored by the Aries Community is the Aries Agent Test Harness (AATH). A group from the BC Gov team initiated work on the AATH. Instead of taking the approach used by the Aries Protocol Test Suite (see the previous section) of the test suite itself being an agent, the AATH is just a test facilitator that talks to two full agents to test their compatibility with each other. The AATH uses a backchannel to each of the agents to execute the tests. This is pictured in the following image. There are two agents being tested and the AATH is using a *"Behavior Driven Development"* engine (currently the Behave engine) to drive the backchannels of both agents to run tests.



The Aries Agent Test Harness
Licensed under CC BY 4.0

With the AATH, two agents are deployed and configuration information from both, along with a test script is passed into the BDD engine. Based on the definition of the script, the BDD engine makes calls (using HTTP) to the backchannels of the two agents, triggering

them to interact with one another using the DIDComm agent-to-agent interface. Results are reported back through the backchannels to the BDD engine and further to the test suite execution report. When needed, the script can trigger the BDD engine to pass data to the agents (via the backchannel) to set the initial state as needed to prepare for the running of a test.

The AATH approach of the test harness using the backchannel controlling the agent-under-test should be familiar to those who have gone through the ACA-Py examples provided throughout this course. For ACA-Py, the backchannel is just an instance of a controller, and is, therefore, extremely easy to implement. It's not surprising that the team responsible for ACA-Py and it's HTTP-centric controller architecture came up with the design for AATH! For agent frameworks that don't provide an HTTP controller interface, a backchannel must be created that takes the backchannel HTTP calls (and responses) from (and to) the BDD engine, and converts them as necessary for the framework agent.

Note: The AATH tests the compatibility of two agents and is not itself an agent (as APTS is). That makes it possible to test the compatibility of any two agents. However, for AATH to be used as an AIP conformance tool, a "reference agent" will need to be agreed to by the Aries community and the other agents will need to be tested for compatibility with the reference agent.

Lab: Aries Agent Test Harness

In this lab we'll look at the Aries Agent Test Harness that can be used to test the interoperability of two Aries agents.

Click here to run the lab.

Getting to One Test Suite

An obvious question to ask at this point is why we have two Aries agent test suites? Ultimately, we will not; one of the two approaches will prove more useful than the other. However, at the time of writing this course, there are pros and cons seen with the two approaches (see the table below), and there is not yet a clear winner. As such, groups in the community are pushing on both approaches until it becomes obvious which one to rally around.

Aries Agent Test Harness (AATH)	Aries Protocol Test Suite (APTS)
The AATH is not itself an agent and so does not have to implement the behavior of an agent. It only has to send out commands to control agents. As such, it *should* be easier to implement. Proponents of the AATH approach think this is the key benefit of the approach.	APTS is an agent (albeit a simplified one) and so must implement all agent behavior. Proponents of the APTS approach think that is not an overwhelming task.
With the AATH approach, two arbitrary agents can be tested against one another for interoperability.	This is not possible with the APTS approach.

For AATH to have a “gold standard” for others to test against, a single agent implementation must be declared as “the one” by the Aries community. It’s not clear how that could (or even should) happen.	The APTS becomes, by definition, the single “gold standard” agent for testing an agent-under-test.
In order to execute negative tests, the agents being tested may need to send invalid messages. Using AATH, it’s not clear how to trigger such behavior without adding code to do the wrong thing into real agents.	Proponents of the APTS approach think this is a key challenge with the AATH approach.

Summary

The young and evolving Aries development tools are just that, young and evolving. Eventually, one test suite will emerge to rule all test suites! For now, it is simply important to understand the features required of an Aries interoperability test suite. We also hope the labs gave you a better idea of how this ultimate test suite will look and behave.

In the next chapter, we will get back into the weeds a bit and delve into Aries mobile agents and message routing.

Chapter 7: Mobile Agents and Message Routing

Chapter Overview

So far in this course we have focused on how a controller injects business logic to control the agent and make it carry out its intended use cases. We have looked at how the controller manages the messages being exchanged by agents and the format and range of the messages that have been defined by the Aries Working Group. In this chapter, we’ll look at the architecture of *mobile* Aries agents. In doing that, we need to look more closely at message routing—how messages get from one agent to another. The routing part is applicable beyond mobile agents, but we’ve combined the two because routing is required from mobile agents. So, even if you aren’t interested in mobile agents, read on, because there are lots of other things to learn in this chapter.

Learning Objectives

In this chapter, you will learn about:

- Agent message routing, particularly the important roles of mediator and relay agents in mobile messaging.
- The concept of agency—a collection of cloud agents that service mobile Aries agents.
- The role of the DIDDoc in message routing.
- Emerging open source mobile agent projects.

Agent Message Routing

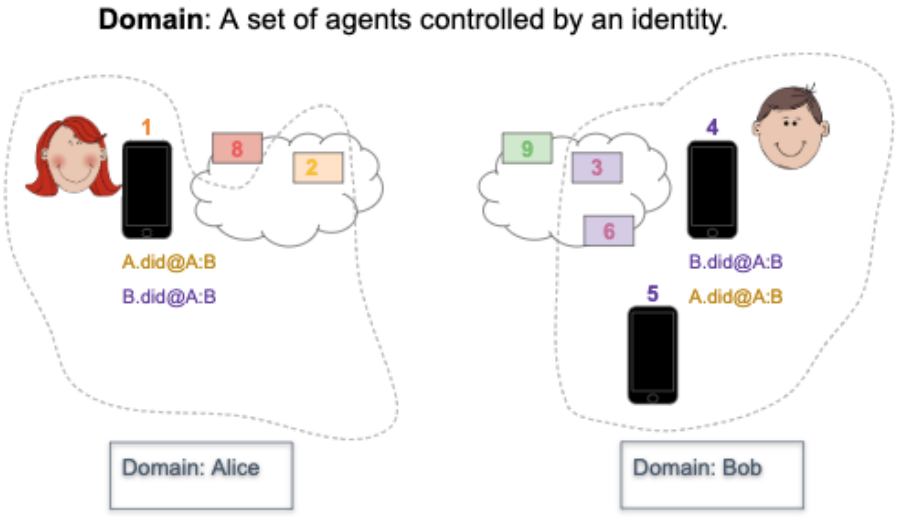
While this chapter is eventually going to be about mobile agents, there is an important digression we have to make in order to understand why mobile agents work the way they do. We'll get into mobile agents soon, but let's first talk about the general topic of Aries message routing—how a message gets from one edge agent to another.

Based on what we have covered in the course so far, it's easy to form a mental model of lots of agents interacting directly with one another; Faber College has its enterprise agent, Alice has her mobile agent app on her smartphone and the two can "directly" message one another whenever the need arises. However, while two agents messaging one another appear to be directly connected, they often are not. Mediator and relay (terms we will formalize shortly) agents are necessary to enable messages to be securely delivered from one edge agent to another because:

- Mobile agents do not have an endpoint (a physical address) that other agents can use for sending messages. Thus, it is impossible for mobile agents and enterprise agents to message each other directly.
- Entities may not want to allow correlation of their agent across relationships and so they use a shared, common endpoint (e.g. https://agents-R-Us.com) such that their messages are "hidden in a crowd" of lots of other messages.
- Entities may not want their inbound messages to be correlated to their outbound messages so they use different paths for sending and receiving messages.
- An enterprise may want to have a single gateway for the use of the many enterprise agents they have in their organization.

Thus, when a DIDComm message is sent from one edge agent to another, it is routed per the instructions of the receiver and for the needs of the sender. For example, using the following picture, Alice might be told by Bob to send messages to his phone (agent 4) via agents 9 and 3, and Alice might always send out messages via agent 2.

In DIDComm, the term **domain** is used to indicate the group of agents that are working for a given entity. Alice's domain has agents 1 and 2, and Bob's agents 3, 4, 5 and 6. Agents 8 and 9 represent **agencies**—service providers that provide domain endpoints, host cloud agents and may provision edge agents on behalf of entities. Our concern in defining domain boundaries is how messages travel from one domain to another. What does Alice have to know to get a message both to Bob's domain (the physical endpoint) and through to Bob's edge agent (the sequence of agents)? Incidentally, there is a Cross-Domain Message RFC (0094) that covers a lot of this material as well.



The Term "Domain" Explained

When Bob and Alice communicate, they want their message to be private—they don't want the intermediate agents to see the contents of their message. So, when there are other agents between the two, Alice and Bob encrypt their messages with wrappers that provide only enough information for each intermediary agent to route (send) the message along on the next step of its end-to-end journey. These wrappers are the equivalent to a postal service envelope with just a "To" address on the outside of the envelope.

To carry the postal system analogy a bit further, suppose Alice and Bob work in different offices of a corporation. Alice might write her message (on paper!) for Bob, put it into an interoffice envelope addressed to Bob, and then in a postal service envelope addressed to the office in which Bob works. She mails the letter via the postal service and it gets delivered to the mailroom at Bob's office. The outer envelope is removed in the mailroom, and the inner envelope is delivered to Bob via the internal mail system. Bob takes the message out of that envelope, and reads Alice's message. This matches the DIDComm world exactly, with Alice using encryption for the envelopes, and the postal service and mailroom as intermediary agents to facilitate delivery.

Mediators and Relays

In DIDComm, the term **mediator** is used for the list of agents Bob provides Alice through which Alice's message will be routed. If there is no list of agents, the message will go directly to Bob, so her agent needs only to encrypt the message for transport to Bob. For each mediator, she explicitly adds another envelope, another layer of encryption and a "To" address. Thus, Bob's "list of agents" is really just a list encryption keys for Alice to use.

Note: As an aside, the folks designing the DIDComm spec have come up with some clever handling to prevent message bloat when there are multiple encryption envelopes. When you repeatedly encrypt the same content, the message can get quite large, and care has been taken to prevent that.

The term **relay** is used in DIDComm to indicate that the message is being routed through one or more additional agents without the knowledge of the sender. The important difference is that the sender must know about all mediators and explicitly add envelope wrappers for each. They don't know (or care) about relays. To stretch our paper message analogy a bit more, the mailroom in Bob's building might deliver all the messages for Bob's floor to an assistant who then distributes the messages to their recipients. Alice doesn't know about that process, and the mailroom handles the "encryption" of putting the messages for Bob's floor into an envelope.

There is (of course!) an Aries RFC that covers mediators and relays in more detail—RFC 0046.

To get back to our Alice and Bob's agents picture, agents 9 and 3 are mediators because Bob explicitly tells Alice, "please send your messages to me through those agents." If Alice is sending her outbound messages via agent 2, then agent 2 is acting as a relay agent for her.

Mediators and Relays are Agents Too!

An important item to underline here is that mediators and relays are themselves Aries agents. They may only do routing activities, or they may do other tasks on behalf of their controlling entity. They have a peer-to-peer relationship with other agents with which they interact, and they use that channel to coordinate the routing they will do on behalf of the edge agent, and to route the messages.

Mediators, Relays and Privacy

When messages pass through other agents there are some privacy implications that need to be considered. It is tempting to simplify the message handling as much as possible, but there are privacy reasons for being careful about how to do that. In this section, we'll give you a taste of the threats to privacy that we are trying to mitigate with DIDComm. Keep these in mind as you think about deploying your DIDComm based services.

Each time a mediator and relay agent receive a message and pass it on, they can record information about the message—metadata. Even though the agent can't see the content of the message, they know at least that the message was sent, how big it is, when it was sent and where it will go next. It is well known that with existing communications systems, the collection of metadata can lead to major invasions of privacy. Telephone metadata (who called who, when and for how long), routinely collected by the telcos, enables the detailed tracking of an individual's social and business relationships. Facebook and Google tracking your logins to other services provides them with information on your interests and frequency of use of that service. ISPs can likewise learn about your interests by tracking the websites you visit from your home computer and mobile phone.

One of the goals in designing the DIDComm messaging protocol was to try to limit the metadata exposed in passing messages. The use of encryption for every wrapper and the minimal inclusion of information in the envelope (just a "To" address for the next step in the route) limits what metadata can be gathered. The use of multiple hops in the sequence

limits the agents from knowing the early and later steps in the flow. From our Alice and Bob picture, let's consider what *could* be collected by the agents:

- Every agent in the flow could try to decrypt the inner messages being passed along.
  – As long as we are using updated, trusted encryption approaches this risk is mitigated. That's covered by DIDComm messaging.
- Agent 9 can see the physical address from which a message came (e.g. from 2). However, it can't see from where agent 2 got the message, so unless all and only Alice's messages come from agent 2's physical IP address, it won't know it's from Alice.
  – This is a reason that Alice would likely not want to run her own agent on her own hardware at home as it would be easier to know when Alice was sending and receiving messages. Enterprises, on the other hand, might not be so concerned.
- Agent 2 can see that messages are going to the physical address of agent 9. Again, as long as people/organizations other than just Bob are using agent 9 as a physical endpoint, agent 2 doesn't know who the ultimate message recipient is.
  – This is the "lost in the crowd" idea. Many people receive their messages at common physical endpoints, and the senders (and other Internet observers) don't learn anything about the ultimate sender or receiver (e.g. Alice or Bob).
- Agent 3 (Bob's mobile agent mediator) knows every time Bob receives a message and when he picks up the messages.
  – This is the same for almost any mobile application operating in conjunction with a cloud service.
- Agent 3 does not know from whom the messages were received. Further, if Bob sends out his messages via a different agent (say agent 6), agent 3 does not know anything about Bob's outbound activities.
  – Of course, if agent 3 and agent 6 are operated by the same service, the service would know about all of Alice's traffic. This leads to the question about how much you trust the service.
  – If agent 9 and agent 3 are operated by the same company, the service would know about all of Alice's inbound traffic, including the source of the messages.
- For a mobile agent, the data sent between the mediator and the mobile agent flows through the mobile service provider that Bob is using. They likewise can know when all the messages are flowing (inbound and out), but cannot decrypt the content of the messages.

Those are at least some of the privacy threats to consider. A security/privacy maven would likely be able to find others. In comparison with the examples we gave (telcos, "login with" services, ISPs), the amount of exposed metadata is less with DIDComm.

In theory, by using different vendors for different agents you could further reduce the metadata each participant can gather. But that comes with its own challenges—for example, the need to engage with each of the different vendors. Other techniques have been suggested. These include:

- Randomly sending “no-op” messages to reduce the knowledge gained about when you send/receive messages
- Adding a chunk of throwaway data to messages to alter the size of the message.

However, these complicate the creation of controllers. It’s all about tradeoffs.

Lab: Using a Mediator

Let’s take a look at a mediator. One of the demos in ACA-Py is the performance script that connects two agents and then executes the issuance of a number of credentials from one agent to the other. An option in that demo is to add a mediator. In this lab we’ll look at how that works.

Click here to get started on the mediator lab.

Mobile Agents and Mobile Agent Mediators

We’re supposed to be talking about mobile agents in this chapter, so why the in-depth look into message routing? While routing is an extremely important thing to understand about DIDComm and Aries agents in general, it is especially important for mobile agents—hence the digression.

As noted in the previous section, mobile agents cannot be directly addressed. All data that gets to any mobile application (agents, games, email—any app) does so via the mobile app making a request to receive the data. You can’t send data to a mobile app, it must request the data from you. As such, Faber’s agent cannot directly send a message to Alice’s mobile agent.

While there is an ability to send notifications to an agent app, the use of notifications is restricted by mobile operating system vendors (e.g. Google and Apple). Only registered services may send notifications to an app, so arbitrary agents can’t message a mobile agent that way. The app stores limit the volume of notifications that can be sent, and require that notifications have an associated message displayed to the user when sent. As such, notifications cannot be used as a way to send messages to an Aries mobile agent.

The bottom line is that in order to operate, a mobile agent **must** have a mediator agent through which all inbound (at minimum) messages must flow. As a result, if you are thinking about creating a mobile agent, you also have to think about how you will deploy a cloud-based mediator agent, how it will be architected and what features it will provide.

Things to Consider When Developing a Mobile Agent

The mediator agent serves other purposes as well. Since mobile agents are not online at all times, and are not constantly polling to see if they have any incoming messages (that consumes resources, particularly data and battery, on the phone), the mediator must provide a queue to hold messages until the mobile agent requests them. The mediator could use the mobile OS (iOS or Android) notification mechanism to let the user know when a message arrives in the queue, triggering a check with the mediator. A mediator could provide backup and restore capabilities for the mobile agent, backing up the agent periodically, and enable a UI supporting the restoration of the agent's storage. The mediator could provide other services as well, but that would come down to trust. How much does the mobile agent owner want to trust the mediator? Let's look at that next.

Mobile Agent Trust

In the previous section we mentioned a few other services that the mediator agent could provide for a mobile agent. The amount that agent could do on our behalf is largely based on how much we can trust the mediator, and the vendor that is providing it. Let's go over what you as a developer of applications in this ecosystem should be thinking about with respect to the trust your clients must have in your products.

As mentioned in the earlier section on privacy, the mediator for a mobile agent will know more than any other about our messaging activity—timing of inbound (at least) messages, when messages are picked up, information about your mobile app (e.g. ISP and other

information to connect with your mobile agent) and your use of any other services it provides. This is the same for any mobile application with an associated web service.

The difference between Aries agents and other mobile apps is that in many cases, Aries agents hold keys that must be tightly held by the owner of those keys. As such, where those keys reside and how they are accessed is paramount in the client trusting the agents. In particular, the private keys that enable Aries protocols such as verifiable credential exchange, should be under the complete control of the agent's owner, and ideally protected by a secure enclave such as is provided on mobile phones. That gives the owner confidence (trust) that the only way to access the agent and all that it is protecting.

A suggestion we often hear from developers new to the community is about making a mobile agent that resides on the cloud as a web service and only put the user interface on the mobile phone. This would be extremely easy to do with ACA-Py since you would be just building the controller on the mobile device. The problem with that approach is that all of the owner's keys would reside on the web service infrastructure and not in the direct control of the owner. The owner would have to trust that the web service would not do anything with the private keys they are holding. That's a big leap compared to having the keys locally, and not one we would recommend.

So, while it is tempting with Aries to build centralized components, as is often effective in other domains, be very careful in doing that. Make sure that the control, particularly of private keys and the use of those private keys is as close to the owner as possible. Using cloud services run by vendors that are just passing along encrypted messages is likely safe enough and necessary. Using cloud services run by vendors that are (for example) actually receiving credentials and proving claims about an entity (a person or an organization) must be considered with skepticism.

Message Encryption Handling

The DIDComm encryption handling is performed within the Aries agent, and not really something a developer building applications using an agent needs to worry about. Further, within an Aries agent, the handling of the encryption is left to libraries—ultimately calling dependencies from Hyperledger Ursa. To encrypt a message, the agent code calls a **pack()** function to handle the encryption, and to decrypt a message, the agent code calls a corresponding **unpack()** function. The "encryption envelope" is described in RFC 0019, including variations for sender authenticated and anonymous encrypting. DIDComm messages are meant to indicate the handling of a message from one agent directly to another, versus the higher level concept of routing a message from an edge agent to a peer edge agent. Much thought has also gone into repudiable and non-repudiable messaging, as described in RFC 0049.

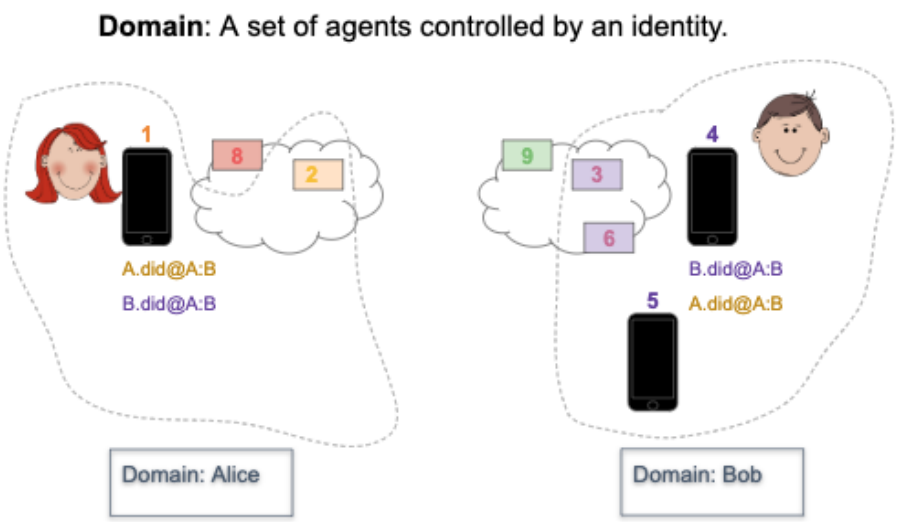
Establishing a Connection with Routing

To this point in the course when we've talked about establishing a connection, we've assumed the two agents are able to talk *directly* to one another. With the scenario described below, mediators are involved in the connection. As you will see, this process

seems to be (Ok, it is) pretty complicated. The details of carrying this out are (at the time of writing) starting to be standardized, so that not all developers will have to figure this all out. Any mobile agent that has a corresponding mediator agent will have implemented this flow already, although likely using an internal/proprietary manner.

The Scenario

We'll use the same Alice and Bob example we used earlier. Here's that picture again.



The Term "Domain" Explained
Licensed under CC BY 4.0

Bob and Alice want to establish a connection so that they can communicate. Bob uses an agency endpoint (https://agents-r-us.com), labelled as 9 and will have a mediator agent, labelled as 3. We'll also focus on Bob's messages to and from his main mobile agent, labelled as 4. We'll ignore Bob's other agents (5 and 6) and we won't worry about Alice's configuration (agents 1, 2 and 8). While the process below is all about Bob, Alice and her agents are doing the same kinds of interactions within her domain.

The Scenario: All the DIDs

A DID and DIDDoc are generated by each participant in each relationship. For Bob's agents (mobile agent and mediator), that includes:

- Bob and Alice
- Bob and his mediator agent
- Bob's mediator agent and agency

That's a lot more than just the Bob and Alice relationship we usually think about!

The Scenario: DIDDoc Data

From a routing perspective, the important information in the DIDDoc is the following (as defined in the DIDDoc Conventions RFC 067):

- The public keys for agents referenced in the routing.
- A service element of type did-communication, including:
  – the one **serviceEndpoint**
  – the **recipientKeys** array of referenced keys for the ultimate target(s) of the message
  – the **routingKeys** array of referenced keys for the mediators

Shown below is an example of these elements of a **service** definition section of a DIDDoc. We saw this in the mediators lab we did earlier. In this example, there is one recipient of the message and one mediator.

```
{
  “service”: [{
    “id”: “did:example:123456789abcdefghi#did-communication”,
    “type”: “did-communication”,
    “priority” : 0,
    “recipientKeys” : [ “did:example:123456789abcdefghi#1” ],
    “routingKeys” : [ “did:example:98490275222” ],
    “serviceEndpoint”: “https://agent.example.com/”
  }]
}
```

Let’s look at the **did-communication** service data in the DIDDocs generated by Bob’s mobile and mediator agents for the set of relationships involved. Recall that there are three relationships, and with two DIDDocs per relationship, we have six(!) DIDDocs to look at:

Bob’s DIDDoc for Alice:	
serviceEndpoint	The endpoint for the agency. This could be empty if the **routingKeys** below contains a public DID for the agency. In that case, the public DID would contain the endpoint.
recipientKeys	Is a key reference for Bob’s mobile agent specifically for Alice.
routingKeys	Key references to the public keys for the agency and mediator agent. A public DID might be used for the agency rather than a public key.

Alice’s DIDDoc for Bob:	
serviceEndpoint	Depends on Alice’s inbound messages configuration.
recipientKeys	Is a key reference for Alice’s mobile agent specifically for Bob.
routingKeys	Depends on Alice’s agent inbound

	messages configuration.

Bob's DIDDoc for his mediator:	
serviceEndpoint	Is empty because Bob's mobile agent has no endpoint. (See the note below for more on this).
recipientKeys	Is a key reference for Bob's mobile agent specifically for the mediator agent.
routingKeys	Is empty.

Bob's mediator's DIDDoc for Bob:	
serviceEndpoint	Is a physical endpoint for Bob's mediator agent.
recipientKeys	Is a key reference for the mediator agent specifically for Bob.
routingKeys	Is empty.

Bob's mediator's DIDDoc for the agency:	
serviceEndpoint	A physical endpoint for Bob's mediator agent.
recipientKeys	A key reference for Bob's mediator agent specifically for the agency.
routingKeys	Is empty.

Agency DIDDoc for Bob's mediator:	
serviceEndpoint	A physical endpoint for the agency.
recipientKeys	A key reference for the agency specifically for Bob's mediator agent.
routingKeys	Is empty.

Note: The null **serviceEndpoint** for Bob's mobile agent is worth a comment. Mobile apps work by sending requests to servers. The server has no way to directly access the mobile app. A DIDComm mechanism called Transport Return Route (RFC 0092) defines how a server can get messages to a mobile agent. It does so by putting the messages into the response to a request from the mobile agent. As well, cloud agents can use mobile platforms' (Apple and Google) notification mechanisms to trigger a user interface event so the person (and app) know there are messages queued.

The Scenario: Preparing Bob's DIDDoc for Alice

Given that background, let's go through the sequence of events and messages that occur when Bob and Alice first connect. Specifically, how Bob's edge agent constructs the service part of the DIDDoc to send to Alice's agent. We'll assume that all of the other connections are already in place—e.g. that Bob's mobile agent has a connection with its mediator and the mediator with the agency. We'll leave how those connections were put in place as an "exercise for the reader."

We'll start the process with Alice sending an out-of-band connection invitation message to Bob, e.g. through a QR code or a link in an email. Here's one possible sequence for creating the DIDDoc. Note that there are other ways this could be done:

- Bob's mobile agent generates a new DID for Alice and prepares and partially completes a DIDDoc, including the public key(s) that he will use when sending messages to Alice.
- Bob messages the mediator agent to send the newly created DID.
  – The mediator agent records the new DID as being associated with Bob.
- Bob's mobile agent fills in the rest of the DIDDoc:
  – The did-communication service endpoint is set to the agency public DID.
  – The recipientKeys array is populated with Bob's new public key for Alice.
  – The routingKeys array is populated with:
    * The public key that Bob has from his mediator agent.
    * The public key (or public DID) that Bob has from the agency.

If there is a public DID used for the agency, Alice will have to resolve (look up on a public ledger) the DID of the agency to get the public key for the agency. Done that way, the agency, which presumably has many users, each with many relationships, can just update its public DID information to, for example, rotate its key, rather than having that done by every user for every relationship they have. The downside of that is that Alice has to regularly check the public ledger for changes to the agency's DIDDoc.

With the DIDDoc ready, Bob uses the path provided in the invitation to send a connection-request message to Alice with the new DID and DIDDoc. Alice now knows how to get any DIDComm message to Bob in a secure, end-to-end encrypted manner. Subsequently, when Alice sends messages to Bob's agent, she uses the information in the DIDDoc to securely send the message to the agency endpoint, from which it is sent to the mediator agent and on to Bob's mobile agent. Now Bob has the information he needs to securely send any DIDComm message to Alice in a secure, end-to-end encrypted manner.

Note: At this time, there are no specific DIDComm protocols for the "set up the routing" messages between the agents in Bob's domain (agency, mediator and mobile agent). Those could be implemented to be proprietary by each agent provider (since it's possible one vendor would write the code for each of those agents), but it's likely those will be specified as open standard DIDComm protocols.

Based on the DIDDoc that Bob has sent Alice and the internal configuration that Alice uses to send messages, the following are the steps that Alice carries out in order to send a DIDComm message to Bob:

- Prepares the message for Bob's agent.
- Encrypts and place that message into a "forward" message for Bob's mediator agent.
- Encrypts and place that message into a "forward" message to Bob's agency endpoint.
- Encrypts and place that message into a "forward" message to Alice's outbound relay.
- Sends that message to her outbound relay.

Note: The first two "forward" messages are required because of what Bob put into his DIDDoc for Alice. The last is independent of what Bob's agent requires and is needed because of how Alice's agent has decided her outbound messages should be handled. Alice could have just skipped that last forward and sent the message straight to Bob's agency endpoint, no relay required.

We've covered a lot in this section and because of the proprietary nature of current implementations, there is no lab. Instead, it's probably worth a second pass through the content to ensure it really sinks in.

Open Source Mobile Agents

It's obvious that particularly for the self-sovereign identity use cases we've been talking about, mobile agents are crucial. Despite their importance, progress on open source agents has been slower, for a couple of reasons. First, many of the organizations working in this space are focused on enterprise solutions (issuers and verifiers), and expect someone else to provide the mobile agents. Second, many of the contributors in the community come from an enterprise background, with less experience in mobile. As well, building and deploying some of the underlying cryptography dependencies has been harder with mobile. The bottom line is that there is less to look at and immediately use in the open source Aries mobile agent space. The good news is that it's changing. Let's look at three emerging open source projects related to mobile agents:

- Open Source Mobile Agent (OSMA)
- Aries React-Native Mobile Agent (ARNIMA)
- React Native Indy SDK (**rn-indy-sdk**)

Open Source Mobile Agent (OSMA)

The Open Source Mobile Agent (OSMA) was initially a combined effort of teams from streetcred.id and Mattr Global. The repo for OSMA can (currently) be found here. The implementation (as of the writing of this section) is based on the predecessor of the **aries-framework-dotnet**, called **agent-framework** (also from streetcred.id). OSMA wraps the framework code using Xamarin, Microsoft's open source development platform for creating

iOS and Android applications. Again, as of writing this section, work is underway by a couple of developers to update OSMA to use the current **aries-framework-dotnet**. Once that is complete, we would expect that the agent will move to be a part of Hyperledger Aries.

While OSMA provides a mobile agent, there is not a cloud mediator agent provided with OSMA. As such, the first challenge a new developer faces in trying to use OSMA is the need to construct a mediator. Several groups have done this, but as yet an open source mediator is not available.

Given the maturity of the underpinnings of OSMA (**aries-framework-dotnet**), this is the most complete open source component for getting started on a mobile agent, with full support for handling connections, receiving credentials and proofing claims.

A stumbling block for some teams getting started with a mobile agent is OSMA's C#/.NET and Xamarin basis. That's a non-starter for some teams looking to use a framework like React Native for mobile development. And that's where our other two open source mobile options come into play.

Aries React-Native Mobile Agent (ARNIMA)

Aries React-Native Mobile Agent (ARNIMA) is a recently announced open-source React Native SDK for building Aries mobile agents. The library was created in the open by AyanWorks, a Sovrin Steward and long time contributor to the Aries community. While the SDK is not as mature as the OSMA base, it does have the advantage (for some) of being built for React Native. As described in the repo's README,

"ARNIMA attempts to meet the needs/asks of many members from the Aries community, mainly React Native developers, who are looking forward to build[ing] cross-platform Aries Mobile Agents using React Native stack."

React Native Indy SDK (rn-indy-sdk)

A third open source project that has just been announced is another React Native library—**rn-indy-sdk**. This project is a contribution from another Sovrin Steward and longtime Aries contributor Absa Bank in South Africa.

The React Native Indy SDK is an implementation of an indy-sdk wrapper for React Native, much like there are for other languages—Python, C#/.Net, Java and so on. While that is a step back from an Aries component, it addresses one of the more challenging parts of building an Aries mobile agent—getting all of the dependencies (particularly the cryptography dependencies) packaged for use in a React Native app. With an Indy SDK in place, the patterns from existing open source Aries projects (such OSMA and even ACA-Py) can be mimicked to build out a full-fledged Aries mobile agent. OK—we admit, it still sounds challenging. But it's far better than starting from scratch!

Lab: Open Source Mobile Agent Projects

In this lab, we'll take a look at the state of open source mobile agents in Aries. As we write this content, there's not too much that can be done, but this is a very quickly evolving part of the Aries world and we'll update the lab as things evolve.

Click here to go to the lab.

Summary

Routing is a pretty intense topic, eh?! There is a lot going on when it comes to Aries mobile agents. The key take-aways from this chapter are:

- Agents are often not directly connected, and in fact, sometimes *cannot* be directly connected (e.g. mobile agents).
- Mediator agents play a key role in Aries agent routing and are required for Aries mobile agent apps. If you are developing mobile apps, you will need to consider how you will deploy your cloud-based mediator agent.
- An agent provides instructions (in the DIDDoc) for how another agent should be routed to it during connection establishment.
- There are several open source mobile agent components that can be used as starting points for building your own mobile agent.

Chapter 8: Planning for Production

Chapter Overview

Now that you have explored building your own mobile agent and/or controller, let's talk about actually getting it into production. This chapter describes some of the things you need to be aware of in the production context, things that are important to the Aries environment. While the vast majority of the content of this chapter will be in the context of enterprise versus mobile agents, we do touch on mobile agent production as well. Pretty cool stuff!

Learning Objectives

In this chapter, you will learn about:

- Mobile agent challenges such as pushing an app to the app store.
- Keeping ledgers and agent storage in sync.
- Writing to a sandbox ledger versus a production one.
- Managing writes to production ledgers, especially those that charge for writes.
- Considerations for the management of your agent, such as backup and restore.
- Horizontal scaling of high-volume, enterprise issuer agents.

Production Challenges—Mobile Agent Apps

The bulk of this chapter will be about production issues around enterprise agents, particularly issuer agents. But we'll first touch briefly on mobile agents and the challenges

they bring.

Note: We are not experts in mobile development so these are based on our observations and experiences only.

The biggest challenge with Aries mobile agents is likely the same with all mobile apps—agent distribution. In particular, there are two issues related to the distribution of new versions:

- Dealing with the app stores' release processes—getting each release of your app through the gates that Apple and Google define.
- Getting users to upgrade to the latest version so that you can drop support for deprecated features in older versions.

For those of us that have grown comfortable (and complacent) in deploying web services that have but a single deployment, having to distribute and upgrade apps on (hopefully) millions of mobile phones creates a much bigger backwards compatibility problem. Sure, with a web service we have to make sure that the web API provides backward compatibility, but that's a more manageable problem. As well, while Google and Apple are aggressive in pushing users to update their apps on a more or less continuous basis, it's still a challenge to keep things working across a range of "stable" releases that users might be running.

Community Upgrade Process

This backwards compatibility challenge is exacerbated in the Aries world because your mobile agents will be integrating with a range of other agents. We talked about the Aries Interop Profile (AIP) in Chapter 5. That's crucial for the community of agent builders.

A second mechanism that is important for managing upgrades in Aries is the "Community Coordinated Update," as described in RFC 0345. This mechanism is used when the Aries community agrees on the need to make a breaking change to the protocols agents are using. We want such changes to be made carefully so that each agent maker can make the changes independently, and all agents will continue to interoperate throughout the transition. With RFC 0345, we have a template process that coordinates a breaking change using a series of steps that gives time for (hopefully) all deployed agents to be updated without breaking agent-to-agent interoperability.

Mobile agents have to be particularly aware of these community coordinated updates because not only do they have to make the changes in their code, but they have to get a new version with that code change distributed to (again, hopefully!) millions of users.

That's all we have for mobile agents. The rest of this chapter is largely focused around challenges with running enterprise agents in production.

Production Challenges—Enterprise Agents

For most of the remainder of the chapter we'll be talking about production issues for enterprise agents. We've had a fair amount of experience in dealing with production use cases and we hope the things that we've learned along the way will make it easier for you. Certainly this is not an exhaustive list, but it will give a good starting point. You'll know more than we did when we got started!

Production Ledger Handling

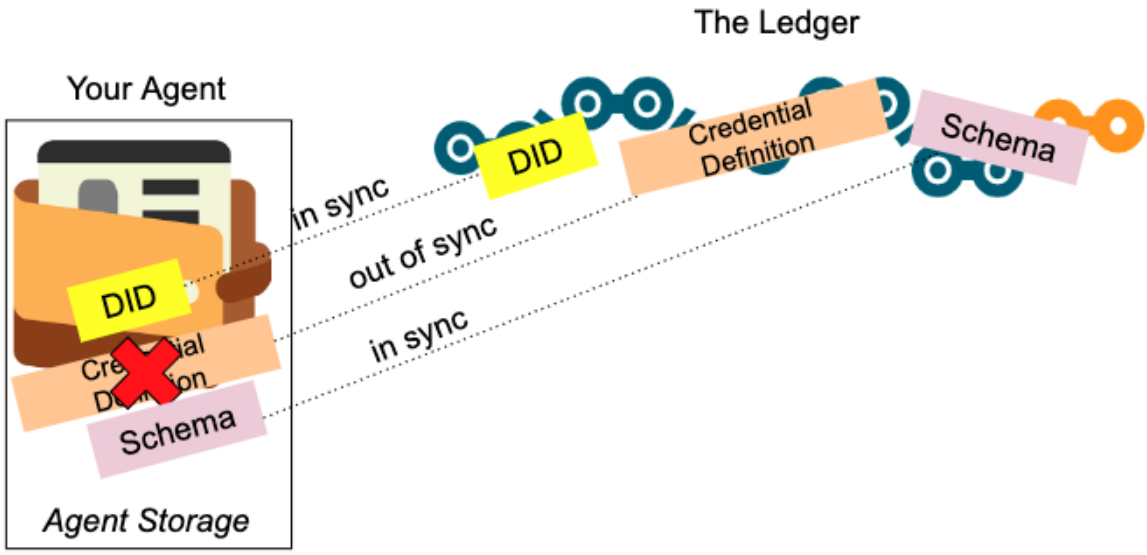
As you develop your first agent and deploy your first proof of concept using a sandbox ledger, you can get quite complacent in managing the ledger. If things aren't working, it's easy to just reset entirely a sandbox ledger, or change the seed so that you can create new objects on the ledger, reset your agent(s) storage and start again. In fact, particularly during development, it's often the case that things unintentionally get out of sync, and you have to delete everything and start again. In production, you don't have that luxury. In some cases (for example, in using the Sovrin production ledger) there is a fee for writing to the ledger. Even more important, if you are an issuer of credentials, you don't want to lose the private keys to anything you have written to the ledger. That means that you have to be extremely careful in managing the agent's persistence—back it up, be able to restore it and always secure it. Remember, your agent's persistence contains the data necessary for your agent to interact with others. You must not let others pretend to be the owner of your agent.

Recreating Ledger Objects (Or Not!)

With Hyperledger Indy, DIDs are created based on a seed that is used to initialize the generating process for the initial public/private key pairs associated with the DID. The use of a seed means if you record the seed and reuse it later, you can recreate the same DID, including the private key. However, the ability to recreate a credential definition from a known seed is not available in Indy. Credential definitions, which contain keypairs per claim in the schema, are not generated from seeds. Once created and published to the ledger,

there is no way to recreate the same credential definition in your agent storage such that you can reuse the credential definition on the ledger.

That means if you lose your agent storage (Indy wallet), you can no longer issue a credential using the credential definition on the ledger.



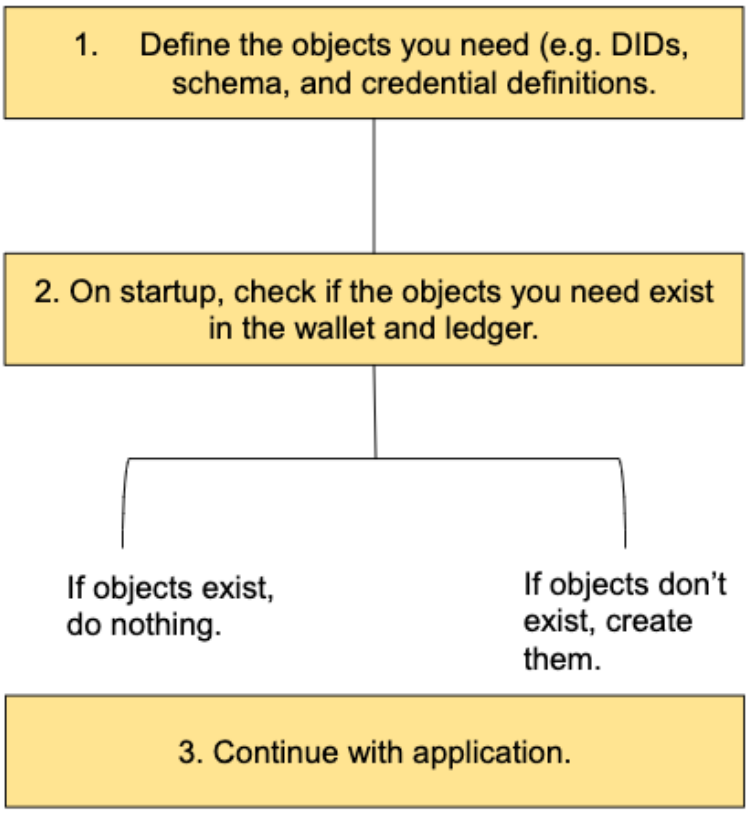
Sync Problem with Recreating Ledger Objects
Licensed under CC BY 4.0

In the previous section, we mentioned that things might “unintentionally get out of sync.” To be a little more precise, that means that the ledger you are using has a credential definition already defined, and the agent storage (Indy wallet) you are using does not. When that happens, you are stuck. You cannot reuse your agent storage (your agent will fail to start) and to move forward, you either have to bump your credential definition version (so a new

version can be written to the ledger), or just reset the ledger. In production, you never want to resort to either of those solutions.

Keeping Development Ledgers and Agent Storage in Sync

Our recommendation for development (and the way we started our proof-of-concept experimenting) was by using the following pattern:



Keeping Development Ledgers and Agent Storage in Sync

At first we used fixed DID seeds and credential definition versions and reset the ledger and agent storage frequently. With fixed DIDs, developers must run a local ledger—since each DID can only be written to the ledger once, to rerun a test, they must first reset the ledger. Later we realized that if we randomly generated a DID seed each time we started up, we could share a ledger and not have to have every developer run a local ledger. In that case, using the pattern above, every time we started a test agent, we followed the "If not, create them" path, since the DID for the agent was always new.

Our feeling is that either pattern works for development, and it's up to the developer to decide which is easier. For each sandbox session you either reset a local ledger and agent storage, or on each start up you create all new objects (based on random DID seeds). But again, sandbox setups and production must be quite different. There are no "resets" in production.

Writing to Sandbox Versus Production Ledgers

In general, Hyperledger Indy sandbox ledgers all use a well-known seed for the DID of a **trustee**, enabling full write access to the ledger. Sandbox applications commonly use the DID of the trustee to create a DID with write access for the agent. The agent's DID is then used to write the additional objects (schema and credential definitions) the agent requires.

In production, the writing of an object is (much) more complicated. On any production ledger, you will not start with access to a seed for a DID that has write privileges. As such, you will need to coordinate with entities that have permission to write (e.g. "endorsers" on Sovrin) to get at least your first DID written. In doing that, your agent may need to accept a "Transaction Author Agreement" (TAA), an agreement much like an End User License Agreement (EULA) that we constantly agree to without reading. As well, you may need to sign the transaction (as the author), and then share it with the endorser to also sign and submit to the ledger on your behalf. In the future, there may be a more automated process where you can pay to write data to the ledger as part of the transaction. Bottom line—it's complicated and still evolving. Managing and containing that complexity is important.

Lab: Scripting Production Writes

In this lab, we'll look at the current process for writing transactions to a ledger that is fully permissioned, such as the Sovrin MainNet.

Click here to run the lab.

Production Ledger Writes

Based on the requirements for operating agent storage in production described in this section and experiences we have had with running in production, let's discuss the current "best practice" pattern for managing ledger objects in production.

Since the creation of objects in production is (more or less) a one-time event, we recommend that the provisioning of ledger objects in production and the ongoing operation of an agent be separate components, as follows:

- The provisioning component makes sure that everything needed exists both in the wallet and on the ledger and if not, creates any objects that are missing. Once that is done, the component exits.
- The operational component makes sure that everything needed exists both in the wallet and on the ledger and if not, exits with an error. If all is well, the component continues in its agent role, but in "read-only" mode, unable to write to the ledger.

Both components operate on the same agent storage (Indy wallet), but by making the operational component "read-only" there is no danger of operational activities causing the ledger to get out of sync with the wallet. Further, on ledgers with fees associated with writing to the ledger (such as the Sovrin MainNet), there is no danger of the operational component unexpectedly writing to the ledger.

Ongoing Writes: Handing Revocations

The tactic outlined in the previous section is relatively obvious when writing an agent that provisions a set of objects on the ledger only at initialization time. It gets less obvious when ongoing writes are needed during agent operations. Let's look at ways of handing those use cases.

When an issuer supports revocation, there will be an ongoing need to write revocation registry updates to the ledger. Revocations could be written as needed (whenever they occur), or written periodically. For example, consider a government that issues driver's licenses as verifiable credentials that needs to revoke existing credentials when driver's license data changes—address, class of license, the right to drive, etc. In that use case, it's likely acceptable to write an update to the revocation registry daily that includes all the revocations that have occurred since the last update. For a large population, the number of daily revocations might number in the thousands. Such an issuer might also define a class of "high importance" revocations (e.g. loss of right to drive) for which the issuer wants to support immediate revocation. On the other hand, issuers that rarely revoke credentials (e.g. monthly) could work either way, revoking credentials as they happen, or batching them into periodic writes to the ledger.

While revocation will be the highest volume use case for ongoing ledger writes, there a couple of other likely use cases:

- An issuer may decide to issue a new type of credential, requiring writing (perhaps) a new schema and a new credential definition, well after initializing their agent.
- An agent may want to rotate the keys on the DIDs they have previously published on a public ledger.
- An agent using Hyperledger Indy may need to create a new revocation registry because they have used up the credentials on the existing registry.
  – In Indy, a revocation registry is created with a defined number of credentials. If the number of credentials issued reaches that number, a new revocation registry must be created in order to issue new credentials.

While we have not yet had much experience with these cases (in fact, just the new schema one), it's likely that the same "provisioning component" approach will work well: have one agent that handles the ongoing ledger writes when needed, and a separate operational agent that assumes writes don't occur. For example, for revocations, an agent instance

could periodically collect the set of revocations from the “system of record” database and perform the necessary ledger write operations, independent of the operational agent.

At this time, we’re not aware of any agent framework that has formalized a separate configuration for handling ledger writes. There would not be anything special about such an agent—the same architecture would be used, just with a provisioning controller that has a very constrained set of capabilities. For example, with ACA-Py, the “ongoing writes” controller would start an instance of ACA-Py that is configured to connect to the same agent storage as the operational issuer ACA-Py instances. The “ongoing writes” agent would execute periodically (perhaps triggered by a cron job) to check what writes to the ledger are needed, perform them, and exit.

Agent Storage Backup and Restore

As was covered extensively in the prerequisite for this course (*LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa*), backup and restore of the agent storage is crucial to the long term usefulness of this technology. While that may be less important in the early days for mobile wallets (users do not write to a public ledger and *should* be able to get credentials easily reissued), backup and restore for enterprise applications is crucial from day one.

Fortunately, unlike the mobile use case, backup and restore for the enterprise use case is a well understood problem. Enterprise agent storage should use an enterprise database, such as PostgreSQL, and all of the tools and techniques that have been developed over the years to manage that enterprise database. There should be nothing special about enterprise agent storage from a database perspective than any other enterprise database. As such, the following guidelines apply:

- Backup the data regularly, either fully or incrementally using logs, depending on the use case.
  – The specific approach taken depends on the answer to the question: how much data can you afford to lose if you lose the operational database and must restore from a backup?
- Test the viability of the backups regularly.
  – Run frequent test restores to verify the integrity of the backup.
- Define and periodically execute a full agent disaster recovery plan.

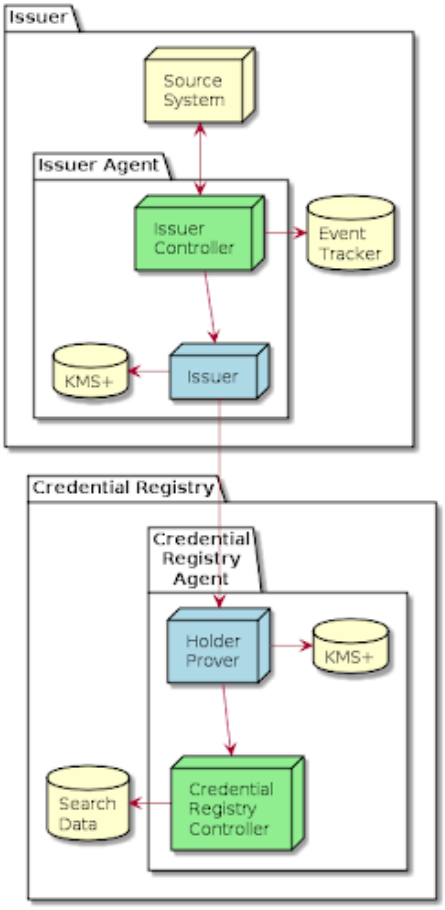
Note: The vast majority of the data in the agent storage database is encrypted and the encryption keys for the database must be appropriately managed. This includes ensuring that the keys are stored separately from the data, they are protected from accidental disclosure, and they are accessible when necessary for starting the agent and after a database restore. Techniques using tools like Hashcorp's Vault for managing the agent storage keys might be a good approach. Again, none of these requirements are different from any other enterprise database system.

Horizontal Scaling

The last enterprise agent production issue we will talk about in this chapter is horizontal scaling—the ability to increase the capacity of an enterprise agent by adding more agent instances. While your use cases may start out small, it's a good idea to consider how you will scale up your capacity as demand grows for the services offered by your agent.

This section draws mainly on the experience of the Verifiable Organizations Network (VON) team at the Government of British Columbia. We won't go into the business case behind OrgBook BC and its code base (you can read about it at the Verifiable Organizations Network website, if you are interested), and we'll just look at the verifiable credential processing requirements.

The VON team has implemented two generations of agents in its OrgBook BC issuer and community holder. For the OrgBook use case, an enterprise issuer must issue millions of credentials to a single enterprise holder as quickly as possible. That sheer volume of credential issue events required the scaling of the solution to use as much processing power as is available. The architecture of the solution is pictured below.



OrgBook BC Issuer-Holder Agents Architecture
Licensed under CC BY 4.0

The following is a summary of the architecture:

- ACA-Py instances are the blue boxes—an issuer and a holder.
- The controllers are in green, each controlling one of the ACA-Py instances.
- The databases labelled “KMS+” are agent storage, holding the keys, ledger data, connections, credentials and protocol state objects.
- The issuer controller monitors the source system for “events” that trigger credential updates—issuing or re-issuing credentials. It invokes the issuer ACA-Py instance to issue the verifiable credentials to the Credential Registries agent using the “Aries Issue Credential” protocol (RFC 0036). The controller keeps track of the events processed and credentials issued in the Event Tracker database.
- The holder ACA-Py instance receives the issued credential, stores it in its agent storage (KMS+) and notifies its controller about the credential.
- The holder controller extracts out the claims from the credentials and stores some of the claims in the search database.
- HTTP is used for transport between all of the controllers and ACA-Py instances.
- All of the databases (both KMS+’s and the Search Database) are all PostgreSQL instances.

The design (hopefully) looks pretty straightforward, and the operation is simple; use Aries protocols to connect two agents and then issue credentials from the issuer to the holder.

The challenge is the volume of credentials to be issued. For the OrgBook BC instance, the requirement was to issue 2.5 million credentials as quickly as possible, ideally within hours. That challenge is somewhat more extreme than might be seen for any high volume issuer. In the more typical case, instead of one holder, the issuer would be connecting with thousands of holders, and issuing credentials to each.

In this scenario, we want to be able to run the agent on a platform that supports (ideally automated) scaling, such as Kubernetes. As the load on the agent increases (more requests), more processes (containers) are added so that we have more available issuing capacity. When requests drop, containers stop and capacity decreases. To enable auto-scaling, cloud native approaches should be applied. In the case of Aries, that means the controllers and agent framework components must be stateless—they must operate without holding state in memory.

As we have seen in this course, both Aries controllers and agent frameworks (such as ACA-Py) operate an event loop (waiting for an event, processing it, and responding to the event), just like any web service. To make Aries components stateless, event state must not be held in memory, but rather persisted to shared storage when completing the processing of an event. With that, any number of instances of the agent components can wait on events, retrieve the state information associated with the event and process it. Since all the transports are HTTP, load balancers can be run in front of the components to distribute the load across all available instances.

Agent frameworks such as ACA-Py have been built with this stateless requirement in mind. As you build your controller, you should also try to meet this requirement. Define a clean controller event loop that includes both retrieving state from shared storage at the start of processing an event and persisting state to that same shared storage at the end of processing. Do not maintain state outside of event processing.

Summary

In this chapter you learned about the challenges of production, both for mobile and enterprise agents. This chapter covered a lot of territory in a short amount of text. You learned about:

- The issue of keeping ledgers and agent storage in sync—and the danger of losing your agent storage (Indy wallet)—not good.
- The “best practice” pattern for managing ledger objects in production.
- The current process for writing transactions to a ledger that is fully permissioned.

Lastly, the BC Government OrgBook architecture was discussed to illustrate how horizontal scaling issues can be addressed in production. Understanding the requirements to enable enterprise agent horizontal scaling and building that in from the start will ensure your agent’s capacity can grow with the load being placed on it.

Chapter 9: What to Do Next

Chapter Overview

We've covered the core of the materials. Congratulations—you've made it! We hope you are well on your way to becoming an Aries developer.

With the heavy content and labs complete, this chapter provides a look forward at what might be next on your journey. We have looked at agents and controllers, agents and protocols and agents and frameworks. We've looked at testing, message routing, mobile agents and moving things into production. Now it's time to consider what you want to do with Aries.

Learning Objectives

In this chapter, you will learn about:

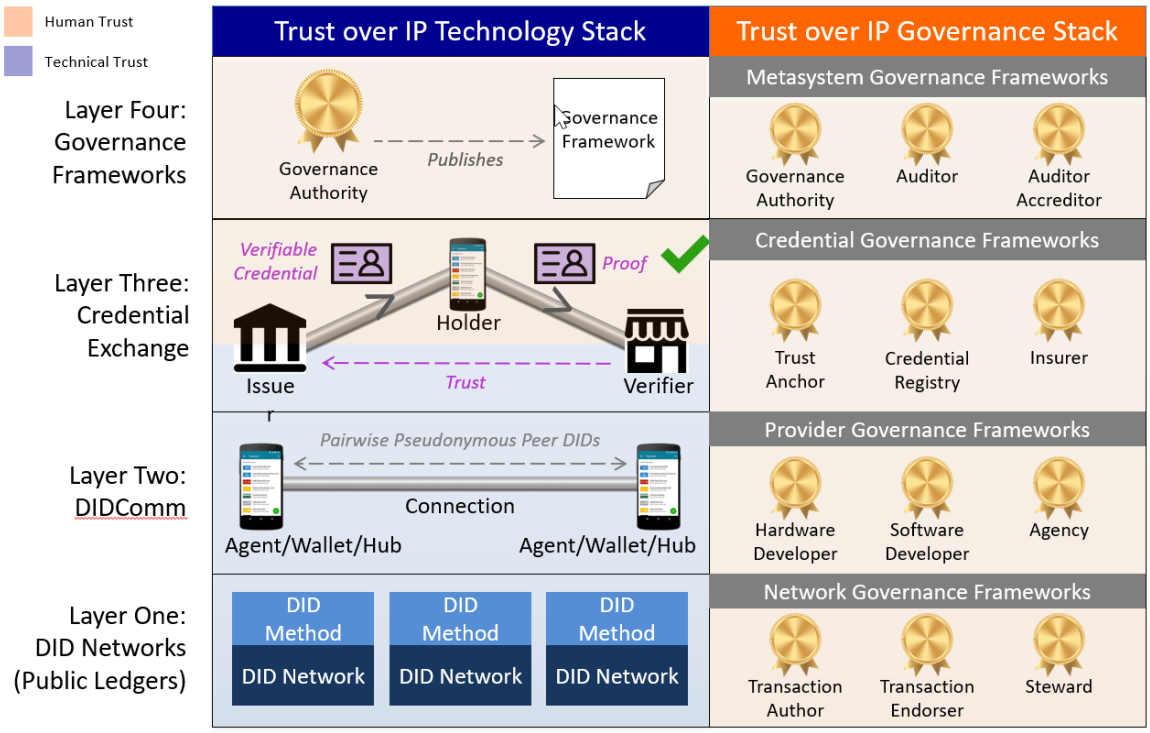
- Where your development efforts might fit best and how they apply to the technical layers of the trust over IP (ToIP) stack.
- What Aries projects are active and where you can contribute.
- Working Call Groups and other ways to get involved.

Where to Go From Here

Going forward, consider what you want to work on next:

- Are you looking to build a product on top of Aries?
- Do you want to add a new capability to Aries?
- Do you want to contribute to the existing Aries projects?
- Do you want to contribute to the projects that are under Aries, such as Indy and Ursa?

In the following, we go through the technical layers of the trust over IP (ToIP) stack from top to bottom and relate that to what you could work on next. As you will recall, the trust over IP stack was introduced in the prerequisite course, *LFS172x – Introduction to Hyperledger Sovereign Identity Blockchain Solutions: Indy, Aries and Ursa* and is represented in this image from Drummond Reed.



Trust over IP (ToIP) Technology Stack
Licensed under CC BY 4.0

Our expectation is that the majority of developers will work on ToIP applications– applications that run on top of Aries agents at Layer Two— there will be fewer contributing developers in the technologies at or below Aries. This is not to dissuade anyone from contributing at the lower levels, but rather to say, if you are not going to contribute at the lower levels, you don't need to know everything about those layers. It's much like web development—you don't need to know TCP/IP to build web apps.

Building Decentralized Identity/Trust Over IP Applications

If you just want to build enterprise applications on top of the decentralized identity-related Hyperledger projects, you can start with building cloud-based controller apps using any language you want, and deploying your code with an agent framework such as **aries-cloudagent-python** (ACA-Py). You can start by using the examples we have provided in the labs in this course, from scratch, or look in the community for starter kits that are beginning to emerge (such as this Verifiable Credential Identity Starter Kit from the BC Government). As we have seen throughout the course, developing enterprise issuer/verifier Aries agents is much like building any web service, receiving events, processing them and responding.

As we covered in Chapter 7, if you want to build a mobile agent, there are open source options available, such as Mattr Global's OSMA, AyanWorks' ARNIMA React Native Aries SDK or ABSA's React Native Indy SDK. You might want to build a general purpose mobile agent that is suitable for use with arbitrary issuers and verifiers. In the long run, we hope that there a number of Aries mobile agents that offer users fantastic mobile experiences. In the short run, we expect that some organizations will want to issue companion mobile agents that work closely with their enterprise web services. Organizations may even

extend their existing mobile applications to include Aries agent/ToIP capabilities. Credit Union tech company CULedger has taken this path with its MemberPass capability that makes verifying the user on support calls way more secure.

As a developer building applications that use/embed Aries agents, you should join the Aries Working Group's weekly calls (see the last section of this chapter) and Aries channels in Hyperledger RocketChat. The maintainers of the Aries repositories often hold regular meetings about their project. For example, ACA-Py holds biweekly ACA-Pug (Users Group) meetings to bring together the ACA-Py developers and teams building applications on ACA-Py. Likewise, the team building **aries-framework-go** holds weekly planning meetings about developments in that repo.

Looking for product ideas? The last chapter of the prerequisite course (LFS172x) provided a list of areas where the verifiable credentials model could be applied, ranging from identity to climate change. Jump back to that chapter to remind yourself of just some of the many possibilities with Aries technology. But realize that the possible applications are truly endless. We hear of new ideas everyday!

As we've talked about a lot in the course, the **aries-rfcs** repo is an important resource. As you get started in building applications, we recommend you carefully review the Aries Interop Profile RFC 0302, where you will find links to the set of RFCs/protocols that many of the existing agents and agent frameworks support. Sticking to these protocols will ensure that your applications will interact with other agent-based applications in the ecosystem. As well, you should look at the state of the agents and agent frameworks that are available, and choose the right one for your application.

Note: If building apps is what you want to do, you don't need to do a deep dive into the Aries agent framework repositories (beyond the API they expose), the **indy-sdk** or the **indy-node** public ledger. You need to know the concepts, but it's not a requirement to know those code bases intimately.

If we did our job right in building this course, you should now have all the tools you need to get started!

Contributing to Aries Projects

As you build applications on top of the Aries projects, you may find limitations in what is available today in Aries frameworks. When that happens the community would love it if you made a contribution. A start would be just to raise the topic on Rocketchat or create an issue in GitHub that clearly outlines what you are trying to do, and the limitation you have hit. From there, the community will be more than willing to help you move forward. Perhaps you haven't yet discovered the capability already exists, or perhaps it's a deficiency that needs to be addressed. Either way, the community will help you find a way to move forward.

The following are some ideas for some of ways you can contribute to Aries projects and the projects on which they are built.

Supporting Additional Ledgers

ACA-Py currently supports only Hyperledger Indy-based public ledgers and verifiable credentials exchange. A goal of Hyperledger Aries is to be ledger-agnostic, supporting other ledgers and verifiable credential models. We're experimenting with adding support for other ledgers, and would very much welcome assistance in doing that. At this time, only the **aries-framework-go** team is actively focused on support for ledgers other than Indy.

Extending Open Source Mobile Apps

We've given three open source mobile "getting started" capabilities in this course. The developers working on those offerings would love to have other contributors with mobile expertise to expand those offerings. Ultimately, all published (in the app stores) mobile agents are by definition proprietary, but there is a lot of shared work needed to make it easier for organizations to create great mobile self-sovereign identity experiences.

Defining and implementing core open source mobile agent mediators is another area currently needing contributions. As we discussed in Chapter 7, developers wanting to deliver mobile capabilities must create a mediator agent through which their mobile agents can connect to other agents. Creating at least an extensible mediator for mobile agents would be a welcome addition to the community.

Mobile Agent Backup and Restore

As we discussed in both this course and its prerequisite, mobile agent backup and restore is a crucial capability that must be both secure and easy for users. Consider designing and building a backup and restore capability for mobile agents that both automates the backup operation (that's pretty easy) and provides a user-friendly, secure restore operation (that's a bit more challenging!).

User Experience

The focus of the Aries (and Indy and Ursa) communities to this point has definitely been on the underlying technology. The majority of the community is technical and the focus has been more on getting things working in a secure manner. However, as products based on the technology begin to move into the mainstream, there is a need to focus on user experience. How can we provide the benefits of this new technology in ways that are easy for the majority of the population? We've seen enough examples in the community to think that this is very much possible. But, ease-of-use needs to be at the core, and as such, we need more great designers to join the community to make that happen.

Driving RFCs from Proposed to Accepted

As we remarked in the discussion on the Aries Interop Profile, only 19 of the RFCs in the **aries-rfcs** repo are referenced in the AIP 1.0. There are a lot of other RFCs that have been proposed and that may be important for building fully capable agents. As you begin to build your applications and discover features you wish you had, ask in the community. You may find that an existing RFC covers your needs . Is your idea new and you need other agents to support it? Raise it in the community and if appropriate, contribute an RFC to drive it with an implementation. In reviewing and contributing RFCs, please make sure you look at the main README for the repo, covering the RFC lifecycle, and the contribution guidance.

Creating a Shared "Aries SDK"

The ACA-Py code base that we've used in many of the labs in the course, as well as some other Aries agent frameworks, embed the **indy-sdk** to provide support for the Hyperledger Indy ledger and verifiable credentials model. The **indy-sdk** also contains features that are independent of the ledger and credentials models, things such as agent-storage (the "Indy wallet") and the cryptography for DIDComm messaging. The Aries community would like to transition from the use of the **indy-sdk** to a more modular structure. The goal is to provide direct support in shared Aries libraries for "agent" capabilities from the **indy-sdk** (e.g. storage), and to provide a common interface for ledger and verifiable credential models to enable support for implementations other than Indy. There are a number of opportunities for developers to contribute at that level.

Improving the Indy SDK

Dropping down a level from Aries and into Hyperledger Indy, the **indy-sdk** needs to continue to evolve. The code base is robust, of high quality and well thought out, but it needs to continue to evolve with new capabilities and improvements to existing features. For example, the Ursa project recently released a new generation of anoncreds (anonymous credentials), the foundation of Indy's ZKP-based verifiable credentials model. The sooner the broader community has access to that code at the Aries level, the better.

As well, a new initiative—called "Rich Schemas"—exists to improve the capabilities of verifiable credentials schema, enabling a common understanding of the content and

meaning of credentials across business domains. This initiative will make it easier for governance groups to define the content and meaning of credentials, making it easier for organizations (governments, businesses, etc.) to consume the credentials issued by others.

The **indy-sdk** is implemented in Rust and generates a C-callable library that can be used by client libraries built in a variety of languages.

Improving indy-node

If you are interested in getting into the public ledger part of Indy, particularly if you are going to be a Sovrin Steward, you should take a deep look into **indy-node**. Like the **indy-sdk**, **indy-node** is robust, of high quality and is well thought out. As the network grows, use cases change and new cryptographic primitives move into the mainstream, and thus, **indy-node** capabilities will need to evolve. **indy-node** is coded in Python. As well, initiatives like the “Rich Schema” project mentioned in the previous section will require work at the **indy-node** level.

Working in Cryptography

Finally, at the deepest level, and core to all of the projects is the cryptography in Hyperledger Ursa. If you are a cryptographer, that’s where you want to be—and we want you there!

How to Get Involved

We’ve covered most of the ways to get involved via links in the content above, so the following is just a list of those resources with links:

- Hyperledger Aries project page
- The Aries Working Group Wiki, and meetings schedule
- Hyperledger Rocketchat and the main Aries channel
- Aries Cloud Agent – Python User Group, meetings schedule, and Rocketchat channel
- Aries Framework Go Wiki, meetings schedule and Rocketchat channel

Starting from those links, you can learn anything more you need about becoming an Aries developer!

Summary

That’s a wrap! Thank you for taking the Becoming an Aries Developer course. We hope that you have acquired a sound understanding of Aries agents and are ready to jump in, contributing to this new and exciting technology.