Mastering Identus: A Developer Handbook

Jon Bauer, Roberto Carvajal

2024-04-05

Table of contents

Welcome

Welcome to the book!

Copyright

License

Book Content:

Applications:

Dedication

Preface

Explains who the authors are and why we were motivated to write this book.

Section I

Introduction

This is a developer-centric book about creating and launching Self-Sovereign applications with Identus. We aim to show the reader how to configure, build and deploy a complex idea from scratch.

This is not a book about Self-Sovereign Identity. There are already great resources available on that topic. If you're new to the idea of SSI or Identus, we recommend the excellent resources listed below as a pre-requisite to this text.

- Self-Sovereign Identity by Alex Preukschat, Drummond Reed, et al. This is the definitive book on SSI and it's ecosystem of topics.
- Identus Documentation

It should be noted that Identus is still new and at the time of writing this book, there are very few best practices, in fact, very little practices at all. A handful of adventurous developers have been building on the platform and sharing their experiences, and we hope to share our own learnings in hopes of magnifying that knowledge and helping developers skip the common pitfalls and bring their ideas to market.

We hope this text will be accessible to anyone who is curious about building decentralized digital identity applications.

We hope that newcomers will be able to use this text to skip common pitfalls and misunderstandings, and bring their ideas to market faster.

We're glad you could join us :)

SSI Basics

Introduction to Self-Sovereign Identity

Self-Sovereign Identity (SSI) represents a paradigm shift in how we manage digital identity. Unlike traditional centralized identity systems where third parties control and store our personal data, SSI puts individuals in control of their own identity information.

Imagine your digital life today: you have dozens of accounts across various websites and services, each with different login credentials and each storing pieces of your personal information. When you apply for a loan, you must provide the same information repeatedly. If a service is breached, your data is exposed. And for over a billion people worldwide without formal identification, accessing basic services remains impossible.

SSI addresses these challenges by creating a decentralized identity layer for the internet where individuals own and control their identity data, sharing only what's necessary with whom they choose.

Core Concepts of SSI

To begin understanding SSI and how it works, these are the fundamental building blocks that you must get familiar with, we will explore in depth each one of these concepts and how to use them in Identus as we progress trough the book.

Decentralized Identifiers (DIDs)

DIDs are globally unique identifiers that enable verifiable, decentralized digital identity. Think of a DID as a digital address that belongs exclusively to you—not assigned by Facebook or Google, but created and controlled by you.

For example, when Alice creates her digital identity in an SSI ecosystem, she generates a DID like `did:prism:4a9bce8d72e4c30017c42f2b` that she controls through cryptographic keys on her smartphone.

DID Documents

A DID Document contains the information payload associated with your DID. It contains public keys for verification and might include service endpoints, for example, with descriptions of ways to communicate with the DID owner, a URL to an asset, etc.

When a bank wants to verify Alice's identity, it can use her DID Document to establish secure communication and verify her credentials without relying on a central authority.

Verifiable Credentials (VCs)

Verifiable Credentials are digital equivalents of physical credentials. Just as your physical wallet might contain a driver's license, university diploma, and health insurance card, your digital wallet contains VCs that prove various aspects of your identity.

For instance, Alice's digital wallet might contain a VC issued by her university confirming her degree, another from her employer verifying her employment, and one from her government confirming her citizenship, all cryptographically secured and under her control.

Verifiable Presentations (VPs)

A Verifiable Presentation allow a holder to interchange cryptographic proof with a verifier about a particular VC. Depending on the format of the Verifiable Credencial, the proof could be about the whole credential or a subset of particular claims. For example, during a regular Verifiable Presentation of a standard W3C JWT VC, the whole credential is presented plus a signed challenge as proof as opposed with a SD-JWT VC, where only a selective disclosure of information from the credential is presented. So when Alice applies for a job, she doesn't need to share her entire identity, just relevant qualifications. Using a VP, she can prove she has a computer science degree without revealing her birth date or address that might also be in that credential.

The Triangle of Trust

The SSI ecosystem operates through what's commonly called the "Triangle of Trust", a relationship between three primary roles in any given interaction. Keep in mind that any person or entity can fulfill each role depending on the context of the interaction, for example, at some point you may receive a credential, this makes you the holder in that interaction, and later that day you may want to verify someone's credential, that would make you a verifier in that interaction. This is one of the first insights that you need to internalize.

Issuer

The entity that issues a Verifiable Credential to a Holder. This could be a bank, government agency or anyone that accepts responsibility for making credentials. For example a governmental agency can issue a passport to a citizen, or a gym can issue a membership to a member. The type of agency is not important, only that they issue a credential. This role accepts responsibility for having issued that credential and should look to establish trust and reputation with Holders and Verifiers.

For example, when the Department of Motor Vehicles issues Alice a digital driver's license, it acts as an Issuer, cryptographically signing a credential that contains her driving privileges and personal information.

The credential includes the DMV's DID in the Issuer field, allowing anyone to verify its origin. The DMV maintains its reputation by issuing accurate credentials and properly verifying identities before issuance.

Holder

The Holder receives and stores credentials. Alice becomes a Holder when she receives her digital driver's license in her wallet app. She controls this credential completely—deciding when, how, and with whom to share it.

When a police officer asks for identification during a traffic stop, Alice can present her digital license through her wallet. The wallet application handles the cryptographic proof that she is indeed the rightful owner of this credential.

Verifier

A Verifier is any person or entity that performs a verification on a Verifiable Credential or any of its referenced entities. A Verifier might perform a check on cryptographic elements of a VC, or make sure that the Issuer DID belongs to the expected entity. Verifiers usually only care to double check that the Verifiable Credential is legitimate and that the included claims meet their expectations.

Following our previous examples, when Alice uses her digital driver's license at a bar, the bartender becomes a Verifier. The bartender's verification app confirms that:

- The credential was issued by a legitimate DMV (by checking the Issuer's DID)
- The credential hasn't been tampered with (through cryptographic verification)
- The credential hasn't been revoked (by checking a revocation registry)
- Alice is the rightful owner (through cryptographic proof)
- Alice is over 21 (by reading the birthdate claim in the credential)

All this happens digitally in seconds, with greater security than visual inspection of a physical ID.

Trust Registry

When Verifiers need to know who a DID belongs to, there needs to be a way to look up that information. A Trust Registry is a mapping between DIDs and the entities they represent. You can think of this like a phone book for DIDs. Trust Registries need to be trusted themselves, and can be as broad or as specific as they need to be. For example, a Real Estate specific Trust Registry that lists real estate agents can be a way to validate that a particular agent is an accepted member of that industry.

In essence, a Trust Registry provides the foundation for knowing which issuers to trust. When the bartender's app verifies Alice's driver's license, it consults a Trust Registry to confirm that the DID belongs to the actual DMV and not an impostor.

For example, a national Trust Registry might list all authorized government agencies and their DIDs, while an industry-specific registry might list accredited universities or professional certification bodies.

DIDs and DID Documents

Overview

A **DID** (*Decentralized Identifier*) is the canonical representation of a DID-Document; a portable, compact hash, which can be passed around easily or stored to a database or blockchain. A DID can be *resolved*, revealing the full, parsable JSON encoded DIDDocument.

A **DID Document** (*Decentralized Identifier Document*) is a JSON-LD (*JavaScript Object Notation for Linked Data*) structure which describes a Subject. This can represent the identity of a person, a thing, or a relationship between one or many entities. Contained in the document is information which can verify that identity without relying on a centralized authority.

TODO: Add DID Method overview A **DID Method** ..

The spec for a `did:prism` DIDDocument can be found here.

An Example DID: `did:prism:4a5b5cf0a513e83b598bbea25cd6196746747f065246f1d3743344b4b81b5a74:Cr`

Let's break down the format of this example DID:

- `did:prism:` The prefix of the DID
- `4a5b5cf0a513e83b598bbea25cd6196746747f065246f1d3743344b4b81b5a74`: The DID identifier. This can be anything, as long as it is unique to the DID Document it is describing, and means something to your application.

- `Cr4BCrsBElsKBmF1dGgwMRJRCglzZWNwMjU...`: The DID Document itself, encoded in base58

An Example DIDDocument:

```
{
  "@context": [
      "https://www.w3.org/ns/did/v1",
      "https://w3id.org/security/suites/jws-2020/v1",
      "https://didcomm.org/messaging/contexts/v2",
      "https://identity.foundation/.well-known/did-configuration/v1"
    ],
  "id": "did:prism:123456789abcdefghi",
  "controller": "did:example:bcehfew7h32f32h7af3",
  "verificationMethod": [{
    "id": "did:prism:123456789abcdefghi#key-1",
    "type": "JsonWebKey2020",
    "controller": ["did:prism:123456789abcdefghi"],
    "publicKeyJwk": {
      "kty": "OKP",
      "crv": "Ed25519",
      "x": "VCpo2LMLhn6iWku8MKvSLg2ZAoC-nlOyPVQaO3FxVeQ"
    }
  }],
  "authentication": ["did:prism:123456789abcdefghi#key-1"],
  "assertionMethod": ["did:prism:123456789abcdefghi#key-1"],
  "keyAgreement": [ "did:prism:123456789abcdefghi#key-1"],
  "service": [{
    "id": "did:prism:123456789abcdefghi#messaging",
    "type": "DIDCommMessaging",
    "serviceEndpoint": "https://example.com/endpoint"
  }]
}
```

Let's look at the components of a DID Document:

- `id`: The DID of the Subject described by the DIDDocument
- `@context`: This is an array of specifications used in this DIDDocument. The first element is usually https://www.w3.org/ns/did/v1 but any other common definitions are JSONWebSignature or DIDComm2 Messaging protocols.
- `controller`: An array of DIDs that are allowed to mutate the DIDDocument
- `verificationMethod`: An array of information which can be used to verify the identity of the Subject.
  - `id`: The DID of the Subject
  - `controller`: The DID of the Subject (author's note: When could this be different than id?)
  - publicKeyJwk or publicKeyMultibase:
    * `publicKeyJwk`: A JSON Web Key (JWK) representation of the Subject's Public Key
    * `publicKeyMultibase`: An encoded public key using Multibase encoding
  - `type`: The type of Verification Method, ie `Ed25519VerificationKey2020` or JsonWebKey2020
- **Authentication Methods**:
  - `authentication`, `assertionMethod`, `keyAgreement`: Arrays of locations in the Subject DID, referenced in a DID + anchor format (`did:prism:1234#authentication0`)
  - *Author's note - Specify these in a more concrete way
- `service`: An array of advertised methods of interacting with the Subject. These could be API endpoints for messaging or file storage systems, but any remote service can be added to add value to the DID.

An non-exhaustive example of a `did:prism` DIDDocument can be found here.

DID Lifecycle

TODO: Describe the lifecycle of DIDs (creation, update, deactivation)

Resolvers

A resolver is a service that can resolve a DID to a DIDDocument. There are PRISM specific resolvers built into Identus SDKs, or you can also run your own resolver service.

Some third-party PRISM resolvers:

- Blocktrust Resolver
- NeoPrism Resolver

Controllers

Controllers are entities that can mutate the DIDDocument. Controllers are specified in the DIDDocument as an array of DIDs so they can be a person, thing, or organization.

Remember that DIDs can all be resolved to DIDDocuments, and each DIDDocument can point to people, things, machines, or services. Every mention of a DID can potentially be a chain of references to other services, or endpoints. There is plenty of room to be creative with this relationship graph.

Identus Concepts

[Identus Application Architecture Diagram]

Identus is made up of several open source components. Each could be used or forked separately but they are designed to work well together.

PRISM Node

PRISM Node implements the `did:prism` method and serves as a second-layer node for the Distributed Ledger, currently it only supports Cardano blockchain or local database but in the future it will be acting as a comprehensive interface to multiple VDR (Verifiable Data Registries). The node can resolve PRISM DIDs and write transactions to a blockchain or database, maintaining an indexed internal state that's synchronized with the underlying blockchain for efficient lookup operations.

As a critical component in the Identus ecosystem, PRISM Node provides a secure and trustworthy platform for storing and managing decentralized identifiers. It handles the creation, update, resolution, and deactivation of PRISM DIDs by generating transactions with the necessary operation information, verifying and validating these operations, and publishing them to the blockchain. Once transactions are confirmed, the node updates its internal state accordingly.

The PRISM Node's architecture enables users to: - Create unpublished DIDs via Apollo building block with the option to announce them publicly later. This means that not all DIDs are required to be published on a VDR, for example, as a Holder you don't beed your DID to be public, but as an Issuer, the Cloud Agent will require the issuing DID to be public and anchored in a VDR. In this case, PRISM Node will handle that operation for you. - Update DID documents by publishing update operations on chain. - Deactivate DIDs by publishing deactivation operations on chain. - Resolving DIDs by querying historical changes on chain. - Track the status of operations submitted to the node.

This second-layer approach is essential for making DIDs scalable and efficient, as it provides the necessary off-chain processing and data storage capabilities while leveraging the security and immutability of the underlying blockchain. PRISM Node is expected to be online at all times to ensure reliable service.

Cloud Agent

Written in Scala, the Cloud Agent runs on a server and communicates with clients and peers via a REST API. It is a critical component of an Identus application, able to manage identity wallets and their associated operations, as well as issue Verifiable Credentials. The Cloud Agent is expected to be online at all times.

The Cloud Agent is designed to be scalable, robust, and standards-compliant, providing comprehensive self-sovereign identity services. It supports W3C standards, DIDCommV2, and Hyperledger Aries protocols, ensuring interoperability within the broader SSI ecosystem. Key capabilities include:

- Support for multiple agent roles including Issuer, Holder, Verifier
- Management of W3C Standard Verifiable Credentials (JSON and JSON-LD formats encoded as JWT), SD-JWT and AnonCreds
- Implementation of DIF Presentation Exchange for credential requests and submissions

- Support for `did:prism` and `did:peer` (version 2) DID methods
- Full implementation of DIDCommV2 messaging and protocols
- Compatibility with Hyperledger Aries RFCs including DID exchange, out-of-band protocol, issue credential, and present proof

The Cloud Agent's REST API enables developers to build controllers in any programming language without needing deep expertise in the underlying SSI standards. This architecture allows business logic to be separated from the identity infrastructure, making it easier to develop specialized applications while leveraging the full power of decentralized identity.

When deployed, the Cloud Agent interacts with the PRISM Node over gRPC protocol, using it as the Verifiable Data Registry to anchor DIDs on a distributed ledger for high security and availability.

Building Blocks

Identus separates the handling of important SSI operations into separate, focused libraries called "building blocks". These modular components can be combined and configured to meet various use cases and product requirements. This modular architecture provides excellent flexibility and customization options, allowing developers to implement decentralized identity solutions tailored to their specific needs.

Apollo - Cryptography

Apollo is a comprehensive cryptographic primitives toolbox that Identus uses to ensure data integrity, authenticity, and confidentiality. It provides the foundation for secure communication and data protection throughout the Identus ecosystem.

Apollo employs cryptographic hash functions to create digital fingerprints of data, allowing the detection of any unauthorized modifications. For example, when a verifier receives a credential, Apollo's cryptographic functions can verify that the credential hasn't been tampered with since it was issued.

Additionally, Apollo implements digital signatures to authenticate the identity of senders and recipients, and uses encryption algorithms to protect sensitive data from unauthorized access. In a healthcare scenario, Apollo's encryption would ensure that patient credentials remain confidential when shared between authorized parties.

Castor - DID

Castor enables the creation, management, and resolution of Decentralized Identifiers (DIDs). It currently supports the native `did:prism` method and the `did:peer` method but the are discussions aligning the team to build a more flexible architecture that would allow anyone to write plugins that could support other DID methods.

When a user creates a new digital identity in an Identus application, Castor generates the DID and associated cryptographic material. For instance, a university issuing student credentials would use Castor to create and manage DIDs for both the institution and potentially for each student, establishing the foundation for trusted digital relationships.

Castor's resolver component can look up a DID and retrieve its associated DID Document, which contains the public keys, authentication mechanisms, and service endpoints needed for secure interactions with that identity.

Pollux - Verifiable Credential

Pollux handles all Verifiable Credential operations, allowing users to issue, manage, and verify credentials in a privacy-preserving manner. This building block is essential for implementing the core functionality of credential exchange in self-sovereign identity systems.

In a real-world employment scenario, a company could use Pollux to issue employee credentials, which employees could then store in their digital wallets. When applying for a loan, an employee could selectively disclose relevant employment information from this credential without revealing unnecessary personal details. The bank could then use Pollux's verification capabilities to confirm the credential's authenticity and validity without contacting the employer directly.

Pollux also manages credential status, enabling issuers to revoke credentials when necessary and allowing verifiers to check if a credential is still valid before accepting it.

Mercury - DIDComm

Mercury provides an interface to the DIDCommV2 protocol, enabling secure, private communication between DIDs regardless of the underlying transport mechanisms. This building block establishes the foundation for all agent-to-agent communications in the Identus ecosystem.

For example, when a citizen wants to share a government-issued credential with a service provider, Mercury facilitates the encrypted, authenticated message exchange between the citizen's wallet and the service provider's verification system. This communication happens peer-to-peer, without requiring centralized intermediaries to facilitate the exchange.

Mercury's transport-agnostic design means that these secure communications can occur over various channels, including HTTP, WebSockets, or

Bluetooth, making it versatile for different deployment scenarios from web applications to mobile devices.

More detailed information on each of the Building Blocks can be found in the Identus Documentation.

Edge Agent

Edge Agents bring agent capabilities to client applications such as websites, mobile apps, and other user-facing software. Unlike Cloud Agents, Edge Agents cannot be assumed to be online at all times, and therefore rely on sending and receiving all communications through an online proxy called a Mediator.

Edge Agents are implemented as SDKs that developers can integrate into their applications, providing a comprehensive suite of SSI functionality directly on the client side. These SDKs handle critical operations including:

- Creating and managing DIDs
- Storing and managing Verifiable Credentials
- Secure communication via DIDComm
- Cryptographic operations for signing and verification
- Local secure storage of identity data

Identus provides Edge Agent SDKs in multiple programming languages to support various platforms:

- **TypeScript SDK**: For web applications
- **Swift SDK**: For iOS applications
- **Kotlin Multiplatform SDK**: For Android applications and cross-platform development

Each SDK implements the same core building blocks (Apollo, Castor, Mercury, Pollux, and Pluto interfaces) as the Cloud Agent, ensuring consistent functionality and interoperability across the entire Identus ecosystem. This architectural consistency allows developers to create seamless experiences where identity operations can be performed either on the client or server side as appropriate for their use case.

Edge Agents typically function as digital wallets, enabling users to maintain control over their credentials and identity data on their personal devices. This approach aligns with the core principles of Self-Sovereign Identity by keeping the user in control of their data and minimizing dependency on centralized services.

Mediator

Mediators act as middlemen between Peer DIDs. In order for any agent to send a message to any other agent, it must know the `to` and `from` DIDs of each message. The sender and recipient together make up a cryptographic connection called a `DIDPair`. Mediators maintain queues of messages for each `DIDPair`. If an Edge Agent is offline, the Mediator will hold incoming messages for them until the agent is back online and able to receive them. Mediators can deliver messages when polled, or push via web sockets. Mediators are expected to be online at all times and be highly available.

Identus officially supports and releases updates for their own Mediator implementation but there are a couple of alternative implementations close to the Identus community. In theory, any DIDCommV2 compatible mediator should work with Identus so this is not an exhaustive list.

- PRISM Mediator
- RootsID Mediator
- Blocktrust Mediator

Both RootsID and Blocktrust provide hosted instances of their mediators that are publicly accessible. While extremely helpful during development, these are not recommended for production Identus deployments as they have no uptime guarantee and will not scale past a small number of concurrent users. We will discuss how to run your own Mediator in Chapter .

Verifiable Data Registry (VDR)

The Verifiable Data Registry (VDR) addresses the critical challenge of ensuring authenticity and integrity for publicly available data in an environment where data is generated and consumed at unprecedented scales. Traditional systems often rely on centralized authorities, introducing single points of failure and undermining trust, especially when multiple independent parties need to rely on the same data.

Key challenges include enabling decentralized verification, guaranteeing data integrity and authenticity without central oversight, ensuring interoperability across diverse storage technologies (databases, blockchains, in-memory caches), and facilitating scalable, trustless collaboration.

The VDR system tackles these issues by providing a unified API and a modular, extensible framework for storing, mutating, retrieving, and removing data. It decouples the application layer from the underlying storage mechanisms through two main pluggable components:

- **Drivers**: These are plugins that implement the actual data storage and retrieval logic. Each driver is tailored to a specific storage backend (e.g., a database, a blockchain, or in-memory storage). Drivers are responsible for executing operations like creating, reading, updating, and deleting data. They also handle the generation and verification of cryptographic proofs (hashes for immutable data, digital signatures for mutable data) to ensure data integrity.

- **URL Managers**: These components construct and resolve URLs that reference the stored data. They embed necessary metadata (like driver identifiers, driver families, and cryptographic proofs) into these URLs using standard query parameters. This allows the VDR system to select the correct driver and verify data integrity when a URL is presented.

By abstracting these operations, the VDR layer ensures consistent and verifiable data operations regardless of where or how the data is stored. It employs cryptographic hashes for immutable data and digital signatures for mutable data, embedded in URLs or retrievable via the driver—to guarantee integrity. This architecture fosters a trustless environment where multiple parties can independently verify data authenticity and integrity, enhancing security and collaboration. The VDR system's design allows for adaptability to various storage backends while maintaining consistent methods for data verification and access.

Section II - Getting Started

Installation - Local Environment

Overview

Hyperledger Identus, previously known as Atala PRISM, is distributed across various repositories. These repositories group together different building blocks to provide the necessary functionality for fulfilling each of the essential roles in Self-Sovereign Identity (SSI), as introduced in Identus Concepts and SSI Basics. Throughout this book, we will detail the setup of each component.

The initial component to set up is our Cloud Agent. This agent is responsible for creating and publishing DID Documents into a Verifiable Data Registry (VDR), issuing Verifiable Credentials, and, depending on the configuration, even providing Identity Wallets to multiple users through a multi-tenancy setup. For now, our focus will be on setting up the Cloud Agent to run locally in development mode, supporting only a single tenant. This step is crucial for learning the basics and getting started. As we progress and build our example application, we will deploy the Cloud Agent in development mode first, then set it up to connect to Cardano `testnet` as our VDR on pre-production mode and, finally, in production mode with multi-tenancy support connected to Cardano `mainnet`. This will be a gradual process as we need to familiarize on each stage of the Cloud Agent.

Identus Releases Overview

Identus is built upon multiple interdependent building blocks, including the Cloud Agent, Wallet SDK, Mediator, and Apollo Crypto Library. To ensure compatibility among these components, it is crucial to identify the correct building block versions that are compatible between them. Identus releases are listed here. The release notes for each vesion provides a compatibility table for each Identus release. This guide will focus on the latest stable Identus release at the time of writing (December 2024).

Pre-requisites

Git

If you're using a UNIX-based system (such as OS X or Linux), you likely already have `git` installed. If not, you can download the installer from Git downloads. Additionally, various GUI clients are available for those who prefer a graphical interface.

To clone the Cloud Agent repository, first go to the Releases page and identify the tagged release corresponding to the Identus release you are targeting (e.g. `cloud-agent-v1.40.0` is part of Identus v2.14 release), then clone the repository with this command:

```
git clone --depth 1 --branch cloud-agent-v1.40.0 https://github.com/hyperled
```

> **Note**
>
> Using `--depth 1` will skip the history of changes in the repository up until the point of the tag, this is optional.

Docker

The Cloud Agent and Mediator are distributed as Docker containers, which is the recommended method for starting and stopping the various components required to run the cloud infrastructure.

To begin, install Docker Desktop, which provides everything you need to get started.

WSL

For Windows users, please refer to How to install Linux on Windows with WSL.

> **Note**
>
> Windows is the least tested environment, the community has already found some issues and workarounds on how to get the Cloud Agent working. We will try to always include instructions regarding this use case.

Before We Run The Agent

Once you have cloned the `identus-cloud-agent` repository and Docker is up and running you can jump right ahead and run the agent, but before we do that if you are not yet familiar with the community projects or the structure of the agent itself, we recommend you to spend a little time exploring the following information, this is optional and you can skip it.

Atala Community Projects

There is a growing list of community repositories that aim to provide some extra functionality, mostly maintained by official developers and community members on their spare time.

Some notable projects are:

- **Pluto Encrypted:** Implementation of Pluto storage engine with encryption support.
- **Identus Store** A secure light-weight and dependency free database wrapper.
- **NeoPrism Resolver:** A did:prism resover and explorer.
- **Blocktrust Mediator** A DIDCommv2 compliant mediator, written in C#.
- **Edge Agent SDK Demos:** Browser and Node versions of Edge Agent SDK integrated with Pluto Encrypted.
- **Identus Test:** Shell script helper that will checkout a particular Identus release and compatible components.

Exploring The Repository

There are two fundamental directories inside the repository if you are an end user.

1. `docs` Where all the latest *technical documentation* will be available, this includes the Architecture Decision Records (ADR), general insights, guides to deploy, examples and tutorials about how to handle VC Schemas, Connections, secrets, etc. We will do our best to explain in detail all this procedures as we build our example app.
2. `infrastructure` This directory holds the agent's Docker file and related scripts to run the agent in different modes such as `dev`, `local` or `multi`. **The way to change the agent setup is by customizing environmental variables trough the Docker file**, so we

> really advise you to get familiar with the `shared` directory content, because that's the base for every other mode, in essence, every mode is a customization of the shared Docker file.

Our first mode to explore and the simplest one should be `local` mode, which by default will run a single agent as a single-tenant, meaning that this instance will control only a single Identity Wallet that will be automatically created and seeded upon the first start of the agent.

The `multi` mode essentially runs 3 different `local` agents but each is assigned a particular role such as `issuer`, `holder` and `verifier`. This is useful in order try test more complex interactions between independent actors.

Finally the `dev` mode is meant to be used for development and provides an easy way to modify the Cloud Agent source code, it does not rely on the pre-built Docker images that the `local` mode fetches and run. We will not use this mode at all trough the book but feel free to explore this option if you would like to make contributions to the Cloud Agent in the future.

Running The Cloud Agent

Environment Variables

Inside `infrastructure/local` directory, you will find three important files, `run.sh`and `stop.sh` scripts and the `.env` file.

Our `local` environment file should look like this

```
AGENT_VERSION=1.40.0
PRISM_NODE_VERSION=2.5.0
VAULT_DEV_ROOT_TOKEN_ID=root
```

This will tell Docker which versions of the Cloud Agent and PRISM Node to run, plus a default value for the `VAULT_DEV_ROOT_TOKEN_ID`, this value corresponds to the HashiCorp Token ID. HashiCorp is a secrets storage engine and it will become relevant later on when we need to prepare the agent to run in `prepod` and `production` modes, for now the `local` mode will ignore this value because by default it will use a local `postgres` database for it's secret storage engine.

The `run.sh` script options:

```
./run.sh --help
Run an instance of the ATALA bulding-block stack locally

Syntax: run.sh [-n/--name NAME|-p/--port PORT|-b/--background|-e/--env|-w/--w
options:
-n/--name              Name of this instance - defaults to dev.
-p/--port              Port to run this instance on - defaults to 80.
-b/--background        Run in docker-compose daemon mode in the background.
-e/--env               Provide your own .env file with versions.
-w/--wait              Wait until all containers are healthy (only in the bac
--network              Specify a docker network to run containers on.
--webhook              Specify webhook URL for agent events
--webhook-api-key      Specify api key to secure webhook if required
--debug                Run additional services for debug using docker-compose
-h/--help              Print this help text.
```

For our first interaction with the agent all we have to do is to call the run script. If you have any conflicts with the port 80 already in use or you don't want that as the default, you can pass `--port 8080` or any other available port that you would like to use.

So, from the root of the repository you can run:

```
./infrastructure/local/run.sh
```

This will take a while the first time as Docker will fetch the required container images and get them running. To check the status of the Cloud Agent you can use `curl` or open a browser window at same endpoint URL (make sure to specify a custom port if you changed it in the previous step, e.g. use `http://localhost:8080`):

```
curl http://localhost/cloud-agent/_system/health
{"version":"1.40.0"}
```

The `version` should match the version of the Cloud Agent defined in the `.env`file.

To stop the agent, you can press `Control + C` or run:

```
./infrastructure/local/stop.sh
```

Congratulations! you have successfully setup the agent in `local` mode. Next we will explore our Docker file in detail and interact with our agent using the REST API.

Agent REST API

The Cloud Agent API

The only way to interact with our newly created Cloud Agent is trough the REST API, this means any action of the Cloud Agent such as establishing connections, creating Credential Schemas, issuing Verifiable Credentials, etc. will be triggered trough the agent API endpoints.

It is crucial to understand that the API is essentially an abstraction of the agent's Identity Wallet. In our initial setup our agent is running on single-tenant mode, therefor it is managing a single default wallet which is assigned the following Entity ID (UUID) `00000000-0000-0000-0000-000000000000`. You can confirm this by running the following command:

```
docker logs local-cloud-agent-1 | grep "default"


... Initializing default wallet.
... Default wallet seed is not provided. New seed will be generated.
... Entity created: Entity(00000000-0000-0000-0000-000000000000,default,00000000-0000-0000-0
```

Later on the book, once we start building our example app, we will setup the agent in multi-tenant mode, meaning that a single agent instance will be capable of managing multiple wallets, we refer to those wallets as custodian wallets. This more advanced setup is useful when your users would want to delegate their Identity Wallet custody to a service instead of managing the wallet themselves.

Besides the `_system/health` endpoint we used earlier to confirm the agent version, there is one more endpoint used to debug runtime metrics:

```
curl http://localhost/cloud-agent/_system/metrics
# HELP jvm_memory_bytes_used
# TYPE jvm_memory_bytes_used gauge
jvm_memory_bytes_used{area="heap",} 1.07522616E8
jvm_memory_bytes_used{area="nonheap",} 1.74044936E8
...
```

This will be useful when debugging a memory or performance issue or when developing.

OpenAPI Specification

The OpenAPI Specification (`OAS`) defines a standard, language-agnostic interface to HTTP APIs. The Cloud Agent API documentation can be found at https://hyperledger.github.io/identus-docs/agent-api/ and besides being very detailed and always updated to the latest, it also comes with the OAS spec yaml file that will allow us to setup `Postman` to easily test our API or use the same OAS standard to auto generate code for client libraries on different language stacks. We will not attempt to repeat this API documentation in the book, rather lets focus on complement the existing documentation and explain with more detail how everything works.

APISIX Gateway

APISIX is in charge of proxying different services inside the container, exposing three routes trough the port you specified to the `run.sh` script (remember it runs on `port 80` by default).

- http://localhost/cloud-agent/ This will be the Cloud Agent API.
- http://localhost/apidocs/ Swagger UI interface test the API.
- http://localhost/didcomm/ Our public DIDCOMM endpoint, this is our communication channel and it's how we send end to end encrypted messages to another peer trough a mediator. We will take a deep dive into DIDCOMM later in the book.

APISIX by default will just expose this services but trough plugins it can be setup as Ingress controller, load balancer, authentication and much more. You can read the APISIX documentation to learn more.

> ⚠ Warning
>
> This is where `CORS` (Cross-Origin Resource Sharing) is setup, be default it will allow any origin, here you can restrict which domains should be allowed to connect to this endpoints. We will revisit this on our customization guide.

Swagger UI

Swagger UI it's a visualization and interactive tool to explore an API. It's automatically generated from your OpenAPI (formerly known as Swagger) Specification and it's often used to support the API documentation.

Our Cloud Agent Docker file includes a container for Swagger UI that is exposed trough APISIX as explained earlier. This means you can use this tool right away after the agent is running.

To use it, just open http://localhost/apidocs/ in your browser and from the server list select:

- `http://localhost/cloud-agent - The local instance of the Cloud Agent behind the APISIX proxy`.

Then, click the `Authorize` button and a small modal window will popup, in there you need to define an `apikey`, even if by default you haven't defined one, this means you can put any value in here, of course later in the book when we set an actual `apikey` you will need to use it here, for now just use `test` as value and it should be fine.

After authorizing, the modal should look like this:

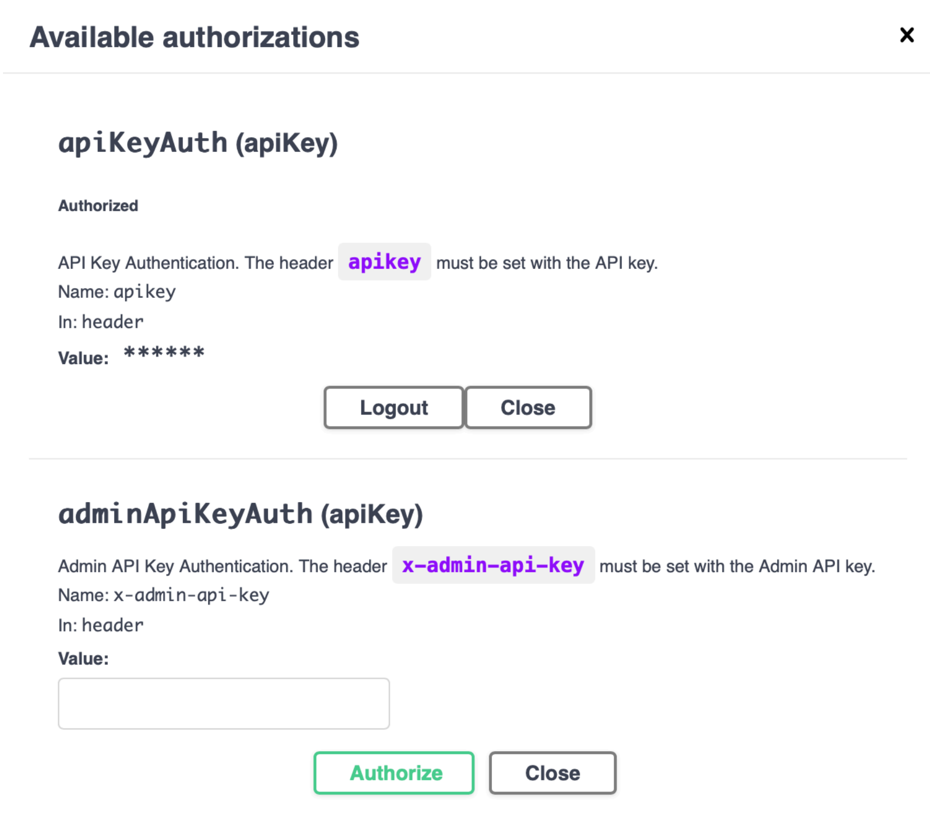

Figure 1: Swagger UI Apikey Modal

You can close that modal window and try your first request, click to expand the `GET /connections` endpoint and click `Try it out` button, that will enable the text inputs for any available parameters, for now, all three parameters should be blank (`offet`, `limit`, `thid`).

Finally, just click the `Execute` button to actually perform the request. This should return something like this:

```
{ "contents": [], "kind": "ConnectionsPage", "self": "", "pageOf": "" }
```


Congratulations! you have connected to the API and asked for a list of `connections`, right now there are no connections so the empty array you get back is correct.

> **Note**
>
> You can use Swagger UI to copy `curl` commands that you can paste in your terminal, this will run exactly the same API request. For example:
>
> ```
> curl -X 'GET' \
>   'http://localhost/cloud-agent/connections' \
>   -H 'accept: application/json' \
>   -H 'apikey: test'
> {"contents":[],"kind":"ConnectionsPage","self":"","pageOf":""}
> ```

Postman

`Postman`is perhaps the most popular API tool among developers, it allow us to easily interact and debug API endpoints but has many killer feature like enabling teams to share and work together on the same API, run automated tests, automatically renew tokens, keeps the state of your interactions with the API, copy code snippets to make API calls over many languages, etc. So it's really a better overall option versus the Swagger UI interface or just directly using `curl`.

The big time saver for us is that because it supports OAS, we can easily import the whole API definition. So, let's try it:

1. If you don't already have it, first you should Download Postman and Sign Up for a free account.
2. Head to the API docs and click the Download button, or copy this direct link `https://hyperledger.github.io/identus-docs/redocusaurus/plugin-redoc-0.yaml`
3. Inside Postman, go to `File -> Import` and either drag & drop your yaml file if you downloaded it or paste the URL in the box, this will auto advance to the next step.
4. On the "How to import" step select "OpenAPI 3.0 with a Postman Collection" and click import.

If everything goes correctly, you should see "Identus Cloud Agent API Reference" in your collections.

Tutorials

The official documentation contains a tutorials section with detailed walkthroughs for each of the most important interactions like connecting to another peer, managing DIDs, managing VC Schemas, issuing a VC, etc. We highly encourage you to follow those and get familiar with the API, this will come in very handy very soon when we start building our own example app.

Mediator

Overview

The `mediator` is a service that enables private and secure message routing between peers. Its primary function is to relay encrypted messages between agents using DIDComm V2 set of communication protocols. Since each participant wallet is Self-Sovereign, mediators help route messages to their intended recipient without any details of each peer besides the agreed public keys over the initial connection protocol, upon which they both create a `did:peer`. Encrypted messages can be sent to peers even if they are not online by relaying on the mediator to manage a queue for the offline party until they are able to connect and retrieve them.

Think of a mediator like a set of Post Office mailboxes. Each mailbox has a mailing address (*a `DID`*). Each mailbox can hold an stack of incoming mail Encrypted DIDComm V2 messages for its owner. Each mailbox also keeps a list of cryptographic Connections with whom its owner can send or receive mail with. In this way, Identus mediators facilitate all types of important transactions, including routing both messages, Verifiable Credential issuance, Credential Exchange or any custom message for your own use.

The key insight for this piece of infrastructure is that each agent can define their own `mediator` to be reachable at, so whenever they establish a `did:peer` with another agent, each peer defines their own reachable `DIDCOMM` endpoint, each one will tell the other their `mediator` as the way to send messages addressed to them.

Why use a Mediator?

Why do we need a `mediator` in the first place? If participants are Self-Sovereign, then why not connect an edge wallet directly to another edge wallet?

SSI participants are connected through various types of devices and network conditions. The Cloud Agent, is expected to be online and available all the time, as it's usually hosted in a cloud infrastructure somewhere. However users who keep their wallets on mobile devices, or laptop computers are only online sporadically and can't be expected to have a reliable connection to the Internet at all times. This is where mediators earn their keep. If `Alice` were to send a message to `Bob`, who happens to be offline at the moment, the message could not be delivered, plus `Alice` would also have to know sensitive data about how to contact `Bob`. If `Bob` changes devices, they would have to tell `Alice`, and all other peer connections. If `Alice` instead, sends the message through the `mediator`, it can be held securely and retrieved the next time `Bob` connects to the Internet. Mediators provide reliability between Connections, without compromising privacy or security.

Set up a Mediator

As usual, we use Docker to setup our infrastructure, `mediator` is no different and here is an example `docker-compose.yml`, `mongo-initdb.js` and `.env` files, you should put all 3 files in a folder.

`docker-compose.yml`

```yaml
services:
  ################
  ### MEDIATOR ###
  ################
```

```
mediator-mongo:
  image: mongo:6.0
  command: [ "--auth" ]
  environment:
    - MONGO_INITDB_ROOT_USERNAME=admin
    - MONGO_INITDB_ROOT_PASSWORD=admin
    - MONGO_INITDB_DATABASE=mediator
  volumes:
    - ./mongo-initdb.js:/docker-entrypoint-initdb.d/initdb.js

mediator:
  image: docker.io/identus/identus-mediator:${MEDIATOR_VERSION:latest}
  ports:
    - "${MEDIATOR_PORT:-8080}:8080"
  environment:
    # Creates the identity:
    # These keys are for demo purpose only for production deployments generate keys
    # Please follow the README file in the Mediator repository for guidelines on How to ge
    # KEY_AGREEMENT KEY_AUTHENTICATION are using format JOSE(JWK) OKP type base64urlsafe e
    - KEY_AGREEMENT_D=${KEY_AGREEMENT_D:-Z6D8LduZgZ6LnrOHPrMTS6uU2u5Btsrk1SGs4fn8M7c}
    - KEY_AGREEMENT_X=${KEY_AGREEMENT_X:-Sr4SkIskjN_VdKTn0zkjYbhGTWArdUNE4j_DmUpnQGw}
    - KEY_AUTHENTICATION_D=${KEY_AUTHENTICATION_D:-INXCnxFEl0atLIIQYruHzGd5sUivMRyQOzu87q
    - KEY_AUTHENTICATION_X=${KEY_AUTHENTICATION_X:-MBjnXZxkMcoQVVL21hahWAw43RuAG-i64ipbeKK
    - SERVICE_ENDPOINTS=${SERVICE_ENDPOINTS:-http://identus-mediator:8080;ws://identus-med
    - MONGODB_USER=admin
    - MONGODB_PASSWORD=admin
    - MONGODB_PROTOCOL=mongodb
    - MONGODB_HOST=mediator-mongo
    - MONGODB_PORT=27017
    - MONGODB_DB_NAME=mediator
  depends_on:
    - "mediator-mongo"
```

mongo-initdb.js

```
db.createUser({
    user: "admin",
    pwd: "admin",
    roles: [
        { role: "readWrite", db: "mediator" }
    ]
});

const database = 'mediator';
const collectionDidAccount = 'user.account';
const collectionMessages = 'messages';
const collectionMessagesSend = 'messages.outbound';

// The current database to use.
use(database);

// Create  collections.
db.createCollection(collectionDidAccount);
db.createCollection(collectionMessages);
db.createCollection(collectionMessagesSend);

//create index
db.getCollection(collectionDidAccount).createIndex({ 'did': 1 }, { unique: tr
// Only enforce uniqueness on non-empty arrays
db.getCollection(collectionDidAccount).createIndex({ 'alias': 1 }, { unique:
db.getCollection(collectionDidAccount).createIndex({ "messagesRef.hash": 1, "

// There are 2 message types  `Mediator`  and `User` Please follow the Readme
const expireAfterSeconds = 7 * 24 * 60 * 60; // 7 day * 24 hours * 60 minutes
db.getCollection(collectionMessages).createIndex(
    { ts: 1 },
    {
```

```
        name: "message-ttl-index",
        partialFilterExpression: { "message_type": "Mediator" },
        expireAfterSeconds: expireAfterSeconds
    }
)
```

`.env`

```
### MEDIATOR
MEDIATOR_PORT=127.0.0.1:8080
MEDIATOR_VERSION=1.0.0
KEY_AGREEMENT_D=xxx
KEY_AGREEMENT_X=xxx
KEY_AUTHENTICATION_D=xxx
KEY_AUTHENTICATION_X=xxx
SERVICE_ENDPOINTS="https://example.com"
```

Your `.env` file will hold all the important bits, lets trough them.

`MEDIATOR_PORT` will define which port the container will be reachable at, I have also bound it to the local interface because I want to proxy this service over SSL with `apache` or `nginx`.

`MEDIATOR_VERSION` will default to `latest` but you can also define a specific version, this is important because sometimes there are breaking changes and your `cloud-agent` or SDK will be compatible with a specific version of the `mediator` that may not be the latest.

`KEY_AGREEMENT_D`, `KEY_AGREEMENT_X`, `KEY_AUTHENTICATION_D`, `KEY_AUTHENTICATION_X` are all the secret keys that are needed for the mediator to generate `DIDs`, in this case the `mediator` generates a `did:peer` so others can route messages trough it. Follow the latest updated guide to generate keys.

`SERVICE_ENDPOINTS` is how you define the *public url* of your `mediator`, this is very important, this is how other agents will reach this service, it can be the public IP or domain, you can include specific ports as well. Make sure `MEDIATOR_PORT` and `SERVICE_ENDPOINTS` are in agreement, for example, if you bound `MEDIATOR_PORT` to the local interface and your `SERVICE_ENDPOINTS` defines a public URL, you need to proxy your local port trough `apache`, `nginx` or similar, if you decide to use expose a public IP, all you need to do is to define both to the same IP and port. This is all contextual of how you run your infrastructure.

Once your done setting the environment you can run it with:

```
docker compose up -d
```

Your mediator should be reachable trough the same endpoint defined in `SERVICE_ENDPOINTS`.

If successful, you should see a simple web page with an invite and a QR code for your `mediator`.

Section III - Building

Example Project

Overview - Airline Ticket Wallet

One of the best ways to learn and understand a software platform is to build something with it. In the following chapters, we will build an example application showing how to use Identus to issue and verify airline tickets.

Users will be able to purchase a flight, and receive a Verifiable Credential representing a ticket and seat assignment. Travelers will be able to present their ticket to Airport Security when requested, and in turn, the security officer will be able to verify the ticket's authenticity.

We will use the Identus Cloud Agent, plus the Typescript SDK in our examples, however the same functionality and source is available in the native language of each Edge Agent SDK; Swift for iOS/Mac and Kotlin for Android. Please see the book's Github page for the complete source code and follow along in your language of choice.

Prerequisites

To follow along with the book, please make sure you have a working Cloud Agent development environment, as described in the Section 2.

Roles

Issuer: Airline

Holder: Traveler

Verifier: Airport Security Officer

Wallets

Overview

Wallets are an essential component on every Self-Sovereign Identity interaction, and as you might have guessed, just like in the physical world where a wallet holds your identifiers (IDs), a digital SSI wallet's function is to store and manage Decentralized Identifiers (DIDs), Verifiable Credentials (VCs), cryptographic keys, and other related assets.

Since many SSI frameworks rely on a Blockchain to publish DIDs, there is a common misconception that SSI wallets work in a similar way. Although both SSI and Blockchain wallets require a `seed phrase` built from a random set of `mnemonics`, thats where the overlap ends, because the balance and history of a Blockchain wallet can be restored from the ledger itself as opposed to an SSI wallet, where all the stored information exists only on the device and needs to be manually backed up and restored.

In essence, a wallet in SSI is piece of software that allows users to store, manage, and present proof of their digital identities and credentials. It acts as a repository for digital assets required to fulfill every SSI interaction, ensuring that users and entities have complete control over their data. In the case of Identus, the Edge Agent SDK provides all the abstractions needed to operate a wallet by an individual on a browser or mobile app, and the Cloud Agent, provides a `REST API` to operate a wallet in the cloud, either to itself (single tenant) or for third-parties (multi tenant) as a custodial wallet.

Edge Agent SDK in Identus

Identus provides it's wallet interface through the Edge Agent SDKs, available in 3 flavors:

- TypeScript for Web and Node apps.
- Swift for iOS and Mac.
- Kotlin for Android and JVM.

Each of the flavors provide the same building block implementations:

- Apollo: Provides a suite of necessary cryptographic operations.
- Castor: Provides a suite of operations to create, manage and resolve decentralized identifiers.
- Pollux: Provides a suite of operations for handling verifiable credentials.
- Mercury: Provides a suite of operations for handling DIDComm V2 messages.
- Pluto: Provides an interface for storage operations in a portable, storage-agnostic manner.
- Agent: A component using all other building blocks, provides basic edge agent capabilities, including implementing DIDComm V2 protocols.

And all of them abstract the usage of each building block through the Agent component.

Most of the time you will be operating the wallet trough the Agent interface unless you require to directly call lower level building block API, for example, you may require to send a custom DIDCOMM message payload format which is not directly supported via the Agent building block, you can still use Mercury directly to achieve that. This approach gives you a simple to use interface trough the Agent but also the flexibility and control that comes with also providing access to the lower level APIs. We highly encourage to dig inside each of the building blocks and study how the

Agent is using them, this will come handy when you need to add custom features to your own software.

There is one building block that is *not* implemented and only provided as an interface, this is Pluto, the storage layer for DIDs, VCs, messages, keys, etc. Identus does not have an opinion on how you should store and retrieve the contents of the wallet, so it's your job to implement this part according to your needs. Fortunately, there is a community project providing one implementation called Pluto Encrypted, this project provides 3 different storage engines: InMemory, IndexDB, LevelDB. As the name suggest, Pluto Encrypted provides full Pluto compatibility plus handles encryption and decryption of the wallet contents, this is very important due to the fact that the wallet stores your DIDs (private keys), VCs, messages and a lot of sensitive information. If you are starting out we highly recommend you to use this implementation before attempting to role your own, it's a great starting place that you can extend and customize to your needs.

Custodial Wallets

In an ideal world, everyone should be willing and able to manage their own identity wallets, this is one of the main characteristics of truly Self-Sovereign ecosystem. In practice, there are many good reasons why an identity wallet would be better managed by a service. Such is the case for companies and entities or even individuals that don't want to deal with the responsibility and risk of self-managing their wallets. For this use case Identus provides the concept of Custodial Wallets. What this really means is that an identity wallet can be managed by the Cloud Agent and used over a REST API. For this particular use case, the Cloud Agent supports a multi-tenant mode in order to onboard and serve multiple identity wallets on the same running instance. We will explain the setup in detail over the production installation section, for now the key insight is that when you access your identity wallet through a Cloud Agent, you are really trusting the storage and management of the private keys of your identity to that service.

Example Project

Swift SDK

TODO: Example code for running an Edge Wallet on Swift SDK

TypeScript SDK

TODO: Example code for running an Edge Wallet on TypeScript SDK

DIDComm

Overview

DIDComm (*Decentralized Identifier Communication*) is a crucial component in the Self-Sovereign Identity (SSI) ecosystem, providing the secure messaging layer that enables Decentralized Identifiers (DIDs) to interact. The current specification, **DIDComm v2**, evolved from earlier iterations to address the growing needs for privacy, security, and interoperability in digital identity communications. Its fundamental purpose is to enable secure, private, and authenticated communication between entities identified by DIDs, regardless of their underlying infrastructure or technology stack, transforming DIDs from static identifiers into dynamic communication endpoints capable of engaging in rich interactions.

Within the SSI technology stack, DIDComm sits above the foundational DID layer (which provides identifiers and cryptographic material) and below the application-specific protocols that define particular interactions like credential exchange. This positioning makes DIDComm the connective tissue of the SSI interactions in Identus, enabling all higher-level identity interactions.

Key Characteristics

DIDComm's design incorporates several distinctive characteristics that make it particularly well-suited for decentralized identity communications:

Transport agnosticism allows DIDComm messages to travel over virtually any communication channel—HTTP(S), WebSockets, Bluetooth, NFC, mesh networks or even QR codes. This flexibility means that identity interactions aren't tied to specific network infrastructures, enabling truly peer-to-peer communications even in offline or intermittent connectivity scenarios.

End-to-end encryption ensures that message contents remain confidential between the sender and intended recipient(s), even when traversing untrusted networks or intermediaries. Using public key cryptography derived from DIDs, DIDComm creates secure communication channels that protect sensitive identity information.

Authentication and trust are inherent in DIDComm, as every message can be cryptographically verified to have originated from a specific DID. This authentication happens without centralized authorities, allowing parties to establish trust directly with one another.

Asynchronous capabilities enable communications that don't require both parties to be online simultaneously. Messages can be queued, stored, and forwarded until they reach their destination, making DIDComm suitable for a wide range of real-world scenarios where continuous connectivity isn't guaranteed.

Message routing and forwarding mechanisms allow communications to traverse complex paths, including through mediators and relays, while maintaining end-to-end security. This capability is essential for entities behind firewalls or NATs, or for preserving additional privacy by obscuring network-level metadata.

Message Structure

DIDComm Messaging messages are structured as encrypted JSON Web

Message (JWM) envelopes. This standardized format ensures interoperability while providing the necessary security properties.

A typical DIDComm message consists of:

- **Headers**: Metadata about the message, including IDs, types, timing information, and routing details
- **Body**: The actual content or payload of the message
- **Attachments**: Optional additional data, which might include structured data, documents, or media

These components are assembled into a JWM, which is then typically encrypted and/or signed according to the security requirements of the interaction. The resulting structure provides a consistent envelope format that can carry any type of content while ensuring its security and integrity.

Each message contains a unique identifier and can reference other messages through threading mechanisms, enabling complex multi-message conversations. Messages also declare their type, which connects them to specific protocols that define the expected sequence and semantics of the interaction.

Protocols

DIDComm defines not just message formats but also a framework for protocols—standardized sequences of messages that accomplish specific tasks. These protocols enable interoperability by ensuring that different implementations understand the same message sequences and semantics.

The protocol framework includes mechanisms for discovering which protocols an agent supports and for versioning protocols as they evolve. This allows for graceful upgrades and backward compatibility in the ecosystem.

Several common protocols have emerged in the SSI ecosystem:

- **Connection establishment protocols** define how two parties can establish a secure DIDComm channel
- **Credential issuance protocols** standardize how verifiable credentials are offered and issued
- **Credential presentation protocols** define how credentials can be requested and presented
- **Trust ping protocols** provide simple mechanisms to verify that a communication channel is active

By standardizing these interaction patterns, DIDComm enables different implementations to work together seamlessly, fostering an open ecosystem rather than siloed solutions.

You can find all currently published protocols in here, as these are maintained by the community, you are also encouraged to build and define your own protocols that fit your particular use case, even if you may not decide to publish them.

Example interaction

To illustrate how DIDComm works in practice, consider a typical credential issuance scenario:

1. An Issuer and Holder establish a DIDComm connection, exchanging DIDs and service endpoints
2. The Issuer sends an offer message indicating which credentials it can provide
3. The Holder responds with a request message for a specific credential
4. The Issuer creates the credential and sends it in an issuance message
5. The Holder acknowledges receipt with an acknowledgment message

Throughout this interaction, all messages are encrypted end-to-end, ensuring that even if they pass through intermediaries, the content remains private between the Issuer and Holder. The messages might travel over different transport mechanisms—perhaps starting with an HTTPS connection but switching to Bluetooth for later messages if the parties come into proximity.

In mediated scenarios, the flow becomes more complex but even more powerful. A Holder might receive messages through a mediator service that provides consistent message delivery even when the Holder's device is offline. The mediator receives encrypted messages, stores them until the Holder connects, and then forwards them without ever being able to read the encrypted contents.

Benefits

DIDComm delivers several crucial benefits that advance the core principles of Self-Sovereign Identity:

Privacy preservation stands at the forefront of DIDComm's design. By encrypting message contents and enabling pseudonymous interactions through pairwise DIDs, DIDComm allows individuals to control exactly what information they share and with whom. The transport-agnostic nature also helps prevent correlation through network metadata.

Decentralized communication frees identity interactions from dependence on centralized messaging providers or identity hubs. Parties can communicate directly or through mediators of their choosing, avoiding vendor lock-in and single points of failure.

Interoperability between different SSI implementations ensures that the ecosystem remains open and competitive. A Holder using one wallet implementation can interact seamlessly with verifiers and issuers using entirely different software stacks, as long as they all support the DIDComm protocols.

User control extends beyond just identity data to encompass the communication channels themselves. Individuals can choose how, when, and where they receive communications related to their digital identity, reinforcing the self-sovereign nature of the system.

DIDComm in Identus

Identus has adopted DIDComm v2 as its standard communication protocol, implementing it throughout the framework to enable secure, private messaging between DIDs. This implementation standardizes many important interactions, including the critical communication between mediators and their peers.

The integration with mediators is particularly important in the Identus architecture. Mediators serve as message handlers that can receive, store, and forward DIDComm messages, enabling asynchronous communication even when recipients are temporarily offline. This capability is essential for practical, real-world identity solutions that must function reliably across diverse connectivity scenarios.

At the moment of writing this chapter Identus supports DIDComm over HTTP endpoints by polling, which is by far the most tested and stable implementation, but also via WebSockets enabled as a feature flag in the Cloud Agent.

For details on how messages are sent and received in Identus via mediators, see Chapter .

Example Project

Swift SDK

TODO: Example code for send/receive DIDCOMM messages on Swift SDK

TypeScript SDK

TODO: Example code for send/receive DIDCOMM messages on Type-Script SDK

Connections

Overview

Now that we have a better understanding of Wallets and DIDs, it's time to embark on our first interaction. In this chapter we are going to explore conceptually what a Connection means in SSI, take a deep dive into DID Peers, explain how they work and why they are needed for secure connections, dissect Out of Band invites and finally hands on example code to achieve connecting edge client to an agent.

Before we move forward we highly recommend to at least read the basic Connection tutorial on the official Identus documentation.

Connections in Self-Sovereign Identity

Connections are fundamental to establishing *trusted interactions* between peers. They enable secure and verifiable communication, allowing entities to exchange credentials and proofs in a decentralized manner. This relationship is established using a specific decentralized identifier standard (**Peer DID**) and is governed by a protocol (**DIDComm**) that ensure the authenticity, integrity, and privacy of the interactions between the connected parties.

There are three roles in an SSI connection:

- **Inviter**: The entity that initiates the connection by sending an invitation.
- **Invitee**: The entity that receives the invitation and responds with a connection request.
- **Mediator**: An intermediary that facilitates message delivery between entities, especially when one or both parties may not always be online.

> **Note**
>
> We will cover mediators in detail later on. For now what you need to understand is that they are used as a service to relay messages between peers, they will store messages and deliver them whenever a peer comes back online, connects to the mediator and fetches their messages.

PeerDIDs

They are a special kind of decentralized identifier with some unique properties that allow them to be perfect for use in order to establish private and secure communications between peers.

DID Documents such as PrismDIDs are meant to be publicly available and resolvable by arbitrary parties, therefor storing them in a VDR such as Cardano blockchain is an excellent way to achieve this requirement in a reliable way.

However, when Alice and Bob want to interact with each other, only two parties care about the details of that connection: Alice and Bob. Instead of arbitrary parties needing to resolve their DIDs, only Alice and Bob do. Thus, PeerDIDs essentially describe a key-pair to encrypt and sign data to and from Alice and Bob, routed trough their preferred mediators, e.g. When Alice accepts an invite from Bob and they engage the connection protocol, Alice generates a PeerDID that allows her to encrypt and sign data routed trough Bob's mediator (mediator Y) that only Bob can decrypt, and vice versa, Bob will generate a PeerDID that allows him to encrypt and sign data routed trough Alice's preferred mediator (mediator X) that only Alice can decrypt.

The key benefits of PeerDIDs are:

1. Decentralized by nature.
2. No transaction cost on blockchain.
3. Private (only the concerned parties know about them).
4. Reusable without any reliance on the internet, with no degradation of trust. (adheres to the principles of local-first and offline-first)

Lets resolve a PeerDID, we call resolve to the unpacking and parsing of a DID in order to read its content and use the DID for the interactions that we need to achieve. For this example we will resolve a PeerDID from Atala's mediator sandbox.

```
curl https://sandbox-mediator.atalaprism.io/did

did:peer:2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvMjRAm7y.Vz6Mkhh1e5CEYYq6JBUcTZ6Cp2ranCW
```

We can see that the mediator returned a PeerDID when we send a `GET` request to the `/did` endpoint. In order to resolve this DID we can use an Universal Resolver website for this example:

```
curl https://dev.uniresolver.io/1.0/identifiers/did:peer:2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsH
```

```json
{
  "@context": "https://w3id.org/did-resolution/v1",
  "didDocument": {
    "@context": [
```

```
      "https://www.w3.org/ns/did/v1",
      "https://w3id.org/security/multikey/v1",
      {
        "@base": "did:peer:2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvMjRAm7
      }
    ],
    "id": "did:peer:2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvMjRAm7y.Vz6Mk
    "verificationMethod": [
      {
        "id": "#key-2",
        "type": "Multikey",
        "controller": "did:peer:2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvM
        "publicKeyMultibase": "z6Mkhh1e5CEYYq6JBUcTZ6Cp2ranCWRrv7Yax3Le4N59R6
      },
      {
        "id": "#key-1",
        "type": "Multikey",
        "controller": "did:peer:2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvM
        "publicKeyMultibase": "z6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvMjRAm
      }
    ],
    "keyAgreement": [
      "#key-1"
    ],
    "authentication": [
      "#key-2"
    ],
    "assertionMethod": [
      "#key-2"
    ],
    "service": [
      {
        "serviceEndpoint": {
```

```
          "uri": "https://sandbox-mediator.atalaprism.io",
          "accept": [
            "didcomm/v2"
          ]
        },
        "type": "DIDCommMessaging",
        "id": "#service"
      },
      {
        "serviceEndpoint": {
          "uri": "wss://sandbox-mediator.atalaprism.io/ws",
          "accept": [
            "didcomm/v2"
          ]
        },
        "type": "DIDCommMessaging",
        "id": "#service-1"
      }
    ]
  },
  "didResolutionMetadata": {
    "contentType": "application/did+ld+json",
    "pattern": "^(did:peer:.+)$",
    "driverUrl": "http://uni-resolver-driver-did-uport:8081/1.0/identifiers/",
    "duration": 4,
    "driverDuration": 4,
    "did": {
      "didString": "did:peer:2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvMjRAm7y.Vz6Mkhh1e5C
      "methodSpecificId": "2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvMjRAm7y.Vz6Mkhh1e5CEY
      "method": "peer"
    }
  },
  "didDocumentMetadata": {}
```

```
}
```

This is what a PeerDID looks like when resolved, please bear in mind that the JSON-LD context for `did-resolution` and extra `didResolutionMetadata` entry are added by the resolver and the actual isolated PeerDID is only what we see inside the `didDocument` payload.

In simple terms, a PeerDID is essentially a JSON payload that contains a set of keys and an optional service endpoints, because this is a mediator PeerDID, it contains service endpoints for `DIDCommMessaging`, in this particular case, you can see it contains two of them, one over regular `https` and other trough `websockets`.

To go deeper in your understanding of PeerDIDs please refer to the full Peer DID Method Specification. In the Hyperledger Identus ecosystem, only PeerDIDs method 2 are supported at the time of this writing.

Out of Band invites

Out of Band (OOB) invites are the entry point for some protocols to take place, they usually are encoded in either a JSON payload or a URL and are distributed "out of band", usually over QR codes, but could be distributed over any medium (Bluetooth, NFC, etc). They gather all required information for one peer to start interacting with another and you can think of them as a way to advertise "coordinates" for anyone that would like to establish an interaction to the inviter.

Following the same example as before, we can see that the sandbox mediator also delivers an OOB invite in a URL form.

```
https://sandbox-mediator.atalaprism.io?_oob=eyJpZCI6ImExNTY4YzEyLTBjZGMtNDY0N
```

If we decode the value of the `_oob` query variable from `base64` we get the `json` payload


```
{
  "id" : "a1568c12-0cdc-4647-9dc6-a5aada6f8247",
  "type" : "https://didcomm.org/out-of-band/2.0/invitation",
  "from" : "did:peer:2.Ez6LSghwSE437wnDE1pt3X6hVDUQzSjsHzinpX3XFvMjRAm7y.Vz6Mkhh1e5CEYYq6JBU
  "body" : {
    "goal_code" : "request-mediate",
    "goal" : "RequestMediate",
    "accept" : [
      "didcomm/v2"
    ]
  },
  "typ" : "application/didcomm-plain+json"
}
```


As you can see, an Out of Band invite is really just a way to package a PeerDID and signaling that it can be used for a particular interaction. In this case, for a `RequestMediate` goal over `DIDComm`.

As a refresher from what we covered, a PeerDID is a way to package a set of keys and optional service endpoints, and so, *because this an OOB invite from a mediator*, this invite has everything you need (a PeerDID and set of service endpoints) to establish this service as your mediator.

Connecting two peers

Now, lets issue another kind of Out of Band invite, one from the cloud agent in order to connect.

The Cloud Agent can generate Out of Band invites, this invite then can be parsed by another peer (say another Cloud Agent or Edge Client) and use it to establish a connection, the end result should be a DID Peer on both sides that allow them to send messages to each other over `DIDComm`.

So, our first step is to generate the invite:

```
curl --location 'http://127.0.0.1:8080/cloud-agent/connections' \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--data '{"label": "test"}'
```

> **Note**
>
> The only parameter we can change when generating an invite is the `label`, this is optional and it is a simple string that you can use to identify the connection later, a good idea would be to use a `uuid` that you generate and manage on your systems, or could be an alias or the reason for the connection, this is all contextual to the interaction and use case so in our case we will go with "test".

The Cloud Agent will respond with a payload that should look like this:


```
{
    "connectionId": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
    "thid": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
    "label": "test",
    "role": "Inviter",
    "state": "InvitationGenerated",
    "invitation": {
        "id": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
        "type": "https://didcomm.org/out-of-band/2.0/invitation",
        "from": "did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7cp
        "invitationUrl": "https://my.domain.com/path?_oob=eyJpZCI6ImZiMzZlZG
    },
    "createdAt": "2025-01-04T12:37:37.059649293Z",
```

```
    "metaRetries": 5,
    "self": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
    "kind": "Connection"
}
```


Once the connection invite is created you can fetch it's details by a `GET`request passing the `connectionId`:

```
curl --location 'http://127.0.0.1:8080/cloud-agent/connections/fb36eddd-d51e-42cf-a6fe-e76d2
--header 'Accept: application/json' \
```

You should get back the same payload as when it was created unless something changed, like the `state`:


```
{
    "connectionId": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
    "thid": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
    "label": "test",
    "role": "Inviter",
    "state": "InvitationGenerated",
    "invitation": {
        "id": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
        "type": "https://didcomm.org/out-of-band/2.0/invitation",
        "from": "did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7cpj.Vz6MkpX7H7SNA6
        "invitationUrl": "https://my.domain.com/path?_oob=eyJpZCI6ImZiMzZlZGRkLWQ1MWUtNDJjZi
    },
    "createdAt": "2025-01-04T12:37:37.059649Z",
    "metaRetries": 5,
    "self": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
    "kind": "Connection"
}
```


Lets dig a bit deeper on what this payload represents.

This invite payload contains some important metadata:

- **connectionId**: The unique identifier of the connection resource, used to fetch the connection details.
- **thid**: The unique identifier of the *thread* this connection record belongs to. The value will identical on both sides of the connection (inviter and invitee).
- **label**: A human readable alias for the connection.
- **role**: The Cloud Agent role on this connection, either `Inviter` or `Invitee`.
- **state**: The current status of this connection, note this is contextual to the Cloud Agent role, so as `Inviter` the states could be: `InvitationGenerated`, `ConnectionRequestReceived`, `ConnectionResponsePending`, `ConnectionResponseSent`. But is also possible for the Cloud Agent to parse someone else's invitation, in that case the Cloud Agent will generate a connection with the `Invitee` role and the possible states for that role are: `InvitationReceived`, `ConnectionRequestPending`, `ConnectionRequestSent`, `ConnectionResponseReceived`.
- **invitation**: The `DIDComm` invitation details.
- **createdAt**: Date and time when this connection was created or received.
- **metaRetries**: The maximum background processing attempts remaining for this record.
- **self**: The reference to the connection resource.
- **kind**: The type of object returned. In this case a `Connection`.

Now lets unpack the `invitation` details.

- **id**: The unique identifier of the invitation. It should be used as parent thread ID (`pthid`) for the Connection Request message that follows.
- **type**: The DIDComm Message Type URI (MTURI) the invitation message complies with.

- **from**: The DID representing the sender to be used by recipients for future interactions.
- **invitationUrl**: The invitation message encoded as a URL.

Lets resolve the `from` DID:

```
curl https://dev.uniresolver.io/1.0/identifiers/did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaE
```

```
{
  "@context": "https://w3id.org/did-resolution/v1",
  "didDocument": {
    "@context": [
      "https://www.w3.org/ns/did/v1",
      "https://w3id.org/security/multikey/v1",
      {
        "@base": "did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7cpj.Vz6MkpX7H7SNA
      }
    ],
    "id": "did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7cpj.Vz6MkpX7H7SNA6ooG5sn
    "verificationMethod": [
      {
        "id": "#key-2",
        "type": "Multikey",
        "controller": "did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7cpj.Vz6MkpX7
        "publicKeyMultibase": "z6MkpX7H7SNA6ooG5snn2MzgyoRadEZtsjNSL1x7HiiLkqyV"
      },
      {
        "id": "#key-1",
        "type": "Multikey",
        "controller": "did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7cpj.Vz6MkpX7
        "publicKeyMultibase": "z6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7cpj"
      }
    ],
    "keyAgreement": [
```

```
      "#key-1"
    ],
    "authentication": [
      "#key-2"
    ],
    "assertionMethod": [
      "#key-2"
    ],
    "service": [
      {
        "serviceEndpoint": {
          "uri": "http://host.docker.internal:8080/didcomm",
          "routingKeys": [],
          "accept": [
            "didcomm/v2"
          ]
        },
        "type": "DIDCommMessaging",
        "id": "#service"
      }
    ]
  },
  "didResolutionMetadata": {
    "contentType": "application/did+ld+json",
    "pattern": "^(did:peer:.+)$",
    "driverUrl": "http://uni-resolver-driver-did-uport:8081/1.0/identifiers/
    "duration": 3,
    "driverDuration": 3,
    "did": {
      "didString": "did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA
      "methodSpecificId": "2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7c
      "method": "peer"
    }
```

```
  },
  "didDocumentMetadata": {}
}
```

Now this looks familiar, we have the usual set of keys and it looks similar to the mediator DID but in this case we see a `serviceEndpoint` that contains accepts `didcomm/v2` and it's intended to be used for `DIDCommMessaging`. What all this means is the Cloud Agent DID is essentially advertising how it will receive `DIDComm` messages. In other words this could be translated as: "Here is an invite to connect to me, on it you will find a DID that has my public keys and a service endpoint where I receive `DIDComm` messages".

The final piece to unpack is the `invitationUrl`, this looks a little odd at first:

```
"https://my.domain.com/path?_oob=eyJpZCI6ImZiMzZlZGRkLWQ1MWUtNDJjZi1hNmZlLWU3NmQyZTYzOGI3MCI
```

The first thing that looks wrong is the domain, where does `my.domain.com` comes from? well, it comes from the Cloud Agent and it's hardcoded, you can't customize it but it really doesn't matter, what matters is the payload of the `_oob` field. The URL is not important as what we really need is inside the `base64` encoded field, lets unpack it.

```
echo 'eyJpZCI6ImZiMzZlZGRkLWQ1MWUtNDJjZi1hNmZlLWU3NmQyZTYzOGI3MCIsInR5cGUiOiJodHRwczovL2RpZC
```

```
{
  "id": "fb36eddd-d51e-42cf-a6fe-e76d2e638b70",
  "type": "https://didcomm.org/out-of-band/2.0/invitation",
  "from": "did:peer:2.Ez6LSgY6Y67mJ75YCZfZYxYEPQJZs3vaEg2Cc91vppoTA7cpj.Vz6MkpX7H7SNA6ooG5s
  "body": {
    "accept": []
  }
}
```

And there it is, the `_oob` encoded payload contains the bare minimum to tell you it's a `DIDComm` invitation from a DID Peer.

Connectionless Credential Presentations and Issuance

In the realm of digital identity, establishing trust and exchanging verifiable credentials typically involves a series of interactions between an issuer or verfier and a holder. While many protocols need an existing connection or relationship, a connectionless flow offers a more streamlined approach for specific scenarios. This method is particularly useful when a prior relationship between the issuer or verfier and the holder has not yet been established or is not necessary for a particular interaction.

The fundamental distinction of connectionless issuance and presentation lies in its initiation. Unlike traditional flows that might require a formal connection setup (e.g., exchanging DIDs and establishing a DIDComm channel) *before* a credential offer can be made, the connectionless flow leverages an Out-of-Band (OOB) invitation to kickstart the process directly.

When an issuer or verifier intends to offer or request proof this way, they generate an OOB invitation. This invitation is essentially a self-contained message or a reference (like a URL or QR code) that the holder can receive through various means (QR code, email, a website, etc). Upon parsing this OOB invitation, the holder's agent can immediately understand the intent to offer or proof a credential.

The key here is that the OOB invitation itself contains enough information, or points to it, for the holder's agent to proceed with requesting the credential offer from the issuer or presenting proof to a verifier. This bypasses the need for a separate, preliminary connection handshake. The issuer or verifier, upon receiving a request derived from this OOB invitation, can then proceed to send the actual credential offer or proof request.

From the holder's acceptance of the offer onwards, the subsequent steps closely mirror those in a standard issuance and proof request protocols. The primary efficiency and distinction of the connectionless approach are concentrated at the beginning of the interaction, enabling a quicker, more direct path when a persistent connection is not a prerequisite. This makes it ideal for scenarios like anonymous attestations, one-time verifications, or public credential offerings where the overhead of establishing and managing connections for every holder is impractical.

Example Project

Cloud Agent

TODO: Instructions on how to create an invite on Cloud Agent.

Swift SDK

TODO: Example code to parse invites on Swift SDK

TypeScript SDK

TODO: Example code to parse invites on TypeScript SDK

Verifiable Credentials

Overview

Verifiable Credentials are an integral part of Self-Sovereign Identity, allowing individuals to keep and control how their personal information is shared.

A Verifiable Credential (VC) is a digital statement made by an **Issuer** about a **Subject**. This statement is cryptographically secured and can be verified by a third party without the need for the **Verifier** to directly contact the **Issuer**. Verifiable Credentials are used to represent information such as identity documents, academic records, professional certifications, and other forms of credentials that traditionally exist in paper form. Coupled with other technology such as Self-Sovereign Identity, Verifiable Credentials can unlock novel and exciting use cases.

Components of a Verifiable Credential:

- **Issuer**: The entity that creates and signs the credential. This could be an organization, institution, government entity or individual.
- **Holder**: The entity or individual to whom the credential is issued to and who can present proof to a **Verifier**.
- **Verifier**: The entity or individual that checks the authenticity and validity of the credential trough requesting proof from a **Holder**.
- **Subject**: The entity or individual about which the claims are made. In many cases, the **Holder** and the **Subject** are the same entity.

- **Claims**: Statements about the **Subject**, such as "Alice has an Educational Credential from Vienna University."
- **Proof**: Cryptographic evidence, using Digital Signatures, that the credential is authentic and has not been tampered with.
- **Metadata**: Additional contextual information which may have content or application specific meaning, like expiration date or credential description.

How Verifiable Credentials Work:

- **Issuance**: The issuer creates a credential containing claims about the subject, the subject DID (public key) and signs it with their private key.
- **Storage**: The holder receives the credential and stores it in a digital wallet.
- **Presentation**: When required, the holder presents the credential to a verifier. Selective Disclosure can be used to reveal only relevant context about a claim, and not all user data.
- **Verification**: The verifier checks the credential's authenticity by validating the issuer's digital signature and ensuring the credential has not been tampered with.

Benefits of Verifiable Credentials:

- **Interoperability**: VCs follow standard formats, making them compatible across different systems and platforms.
- **Privacy**: Holders can share only the necessary information, protecting their privacy.
- **Security**: Cryptographic techniques ensure the integrity and authenticity of credentials.
- **Decentralization**: VCs do not rely on a central authority for verification, reducing single points of failure.

Use Cases:

- **Digital Identity**: Proof of identity for accessing services.
- **Education**: Digital diplomas and certificates.
- **Healthcare**: Vaccination records and medical certificates.
- **Employment**: Professional qualifications and work experience.

Formats

There are several formats for Verifiable Credentials, including:

- W3C - v1.1
- W3C - v2.0 (*Atala Roadmap*)
- SD-JWT VC - Active Internet-Draft
- OID4VCI (*Atala Roadmap?*)
- AnonCreds

Schemas

Issuing a Verifiable Credential (VC) requires a credential schema, which serves as a general template defining the valid claims (attributes) the VC can contain. This schema acts as a reference point to ensure that the VC is correctly formatted and valid by checking its claims against the predefined structure.

Schemas can optionally be published on a Verifiable Data Registry (VDR), which is particularly beneficial for widely applicable schema types. Publishing schemas on a VDR facilitates their adoption by other parties, enabling third parties to issue VCs that conform to the same standardized credential format.

For example, relevant entities or industry consortiums can collaboratively develop, agree upon, and publish schemas that they will adopt and recognize. This approach encourages other players in the ecosystem to adopt these schemas as well, fostering interoperability and growth within the ecosystem.

By standardizing schemas, the VC ecosystem becomes more cohesive and efficient, allowing for easier verification and broader acceptance of credentials across different platforms and organizations.

Example Credential Schema

```
{
  "$id": "https://example.com/driving-license-1.0",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "description": "Driving License",
  "type": "object",
  "properties": {
    "emailAddress": {
      "type": "string",
      "format": "email"
    },
    "givenName": {
      "type": "string"
    },
    "familyName": {
      "type": "string"
    },
    "dateOfIssuance": {
      "type": "string",
      "format": "date-time"
    },
    "drivingLicenseID": {
      "type": "string"
    },
    "drivingClass": {
      "type": "integer"
```

```
    }
  },
  "required": [
    "emailAddress",
    "familyName",
    "dateOfIssuance",
    "drivingLicenseID",
    "drivingClass"
  ],
  "additionalProperties": true
}
```

- Publishing your Schema (Milestone 3)

Issuing

Issuing a Verifiable Credential (VC) is a multi-step process that occurs between an issuer agent and a holder. Currently, this process is only supported through the cloud agent's API endpoints, as there is no functionality to issue VCs from edge client SDKs.

The issuing process shares a common prerequisite across all three supported VC formats (JWT, SD-JWT, and AnonCreds): an established connection between the issuing cloud agent and a holder. The holder can be either another cloud agent or an edge client device.

Additional requirements vary depending on the VC format:

1. For JWT and SD-JWT:
   - The issuing agent must have a published DID Prism.
2. For SD-JWT only:

- The holder must also have a DID Prism, but it doesn't need to be published on-chain.

3. For AnonCreds:

   - No additional requirements beyond the established connection.

Issuer flow

From the issuer perspective this is the regular flow to issue a VC:

1. Create a credential offer over API endpoint
2. Send the credential offer to holder over DIDCOMM
3. Receive credential request from holder over DIDCOMM
4. Issue and process credential
5. Send credential to holder over DIDCOMM

Depending on the value of `automaticIssuance` the credential will be automatically issued on step 4 as soon as the credential request is received from the holder, if `automaticIssuance` is set to `false`, the issuer must manually trigger issuance and process trough an API call.

Holder flow

From the holder perspective this is regular flow to receive a VC:

1. Received offer over DIDCOMM
2. Accept offer, in this step the SDK calls cloud agent API endpoint to trigger the issuance of the VC
3. Receive credential over DIDCOMM

TODO: Add code / API calls on Milestone 3.

Revoking

Revoking a VC is done through a simple API call to the cloud agent to revoke a specific credential by it's ID. The end result is that any presentation proof request will fail if the VC has been revoked.

TODO: Add code / API calls on Milestone 3.

Verification

Presenting proof

Verification is a crucial interaction between Self-Sovereign Identity (SSI) agents. This process involves a `verifier` creating a present proof request to a `holder`, who then responds with a proof presentation. The verifier agent subsequently verifies this presentation.

The verification process consists of several key steps:

1. The verifier initiates a present proof request.
2. The holder responds with a proof presentation.
3. The verifier agent automatically verifies the presentation's cryptographic integrity.
4. The verifier makes a final decision to accept or reject the Verifiable Credential (VC).

It's important to note that a valid and correctly formatted VC alone is insufficient for acceptance. While the verifier agent automatically checks the cryptography, the verifier must also:

1. Confirm the trustworthiness of the issuer.
2. Evaluate and accept the claims within the VC.

Thus, the verifier has the final say in accepting or rejecting the verified proof presentation.

The core of the verification process involves the holder signing a random challenge along with the presentation. The verifier agent extracts the public key from this signature and compares it with the public key of the VC's `subject`. If these public keys match, it confirms that the entity responding to the proof presentation request has access to the private key of the VC's subject.

Presentation policies

To streamline the verification process, verifier cloud agents can implement `presentation policies`. These policies serve as an automated mechanism to enhance efficiency and consistency in verifying Verifiable Credentials (VCs).

Key aspects of presentation policies include:

1. Trusted Issuer Lists:
    - Policies establish sets of `trusted issuers` for specific VC `schemas`.
    - This allows verifiers to pre-approve credible sources for particular types of credentials.
2. Dynamic Management:
    - Policies can be updated and managed over time.
    - Verifiers can add or remove trusted issuers as needed, adapting to changes in the ecosystem or their own requirements.
3. Automated Rejection:
    - The verifier cloud agent can automatically reject a presentation proof if the issuer is not listed in the relevant presentation policy.
    - This reduces manual intervention and speeds up the verification process for non-compliant credentials.

4. Schema-Specific Trust:
    - Policies are tied to specific VC schemas, allowing for granular control over which issuers are trusted for different types of credentials.

By implementing presentation policies, verifiers can significantly reduce the manual effort required in the verification process, while maintaining control over which issuers they deem trustworthy for different types of credentials. This approach balances automation with the flexibility to adapt to changing trust relationships in the Self-Sovereign Identity ecosystem.

Selective disclosure

Standard Verifiable Credentials (VCs) presented in JWT format require holders to disclose all content for verifiers to validate integrity and confirm authenticity. However, this approach can compromise privacy when verifiers only need to check specific claims.

Consider a common scenario: age verification at a bar. While establishments typically only need to confirm if a customer is of legal drinking age (18 or 21 in many jurisdictions), traditional ID checks reveal excessive personal information, including full name, address, and exact date of birth. This over-disclosure becomes particularly problematic in digital interactions, where shared information can be easily copied and archived across the internet.

To address this privacy concern, the concept of `selective disclosure` was developed. This approach utilizes `zero knowledge proofs`, a cryptographic technique allowing the sharing of specific information subsets without revealing the entire content.

Hyperledger Identus currently supports two VC formats that enable selective disclosure proofs:

1. SD-JWT (Selective Disclosure JSON Web Token)
2. AnonCreds (Anonymous Credentials)

These formats allow holders to prove specific claims (e.g., being over 21) without disclosing unnecessary personal details, striking a balance between verification needs and privacy protection.

Some of the benefits of Selective Disclosure include:

1. Enhanced Privacy: Minimizes the exposure of sensitive personal information.
2. Compliance: Aligns with data protection regulations by adhering to data minimization principles.
3. User Control: Empowers individuals to manage their digital identity more effectively.
4. Reduced Risk: Limits the potential for data breaches or misuse of personal information.

TODO: Add API request and code examples (Milestone 3)

Section IV - Deploy

Installation - Production Environment

Overview

A production environment setup requires connecting Hyperledger Identus to the Cardano blockchain as the Verifiable Data Registry (VDR). This is achieved through the `prism-node` component, which abstracts the VDR operations for publishing, resolving, updating, and deactivating Decentralized Identifiers (DIDs).

According to the official documentation:

> The PRISM Node generates a transaction with information about the DID operation and verifies and validates the DID operation before publishing it to the blockchain. Once the transaction gets confirmed on the blockchain, the PRISM Node updates its internal state to reflect the changes.

While our local setup instructs `prism-node` to use a local database as the VDR for testing and development, it lacks the benefits of Cardano's secure and decentralized blockchain for publishing DIDs on-chain. To reach the blockchain, our `prism-node` needs to connect to two components:

1. A full node Cardano wallet to submit transactions to the blockchain
2. Cardano-DB-Sync to read the blockchain through a normalized database interface

When the cloud agent needs to publish, update, or deactivate DIDs, it requests `prism-node` to create and validate a transaction, which is then passed to the `cardano-wallet` for submission to the blockchain. Simultaneously, `prism-node` connects to `cardano-db-sync` to read new blocks, filter DID Prisms published on-chain, and notify the cloud agent when the DID operation reaches a certain number of confirmations.

Production and pre-production installations are similar, with the main difference being that production points to `mainnet` and pre-production to `testnet` for both `cardano-wallet` and `cardano-db-sync`. This is achieved by changing environmental variables passed to the Docker containers.

For a production setup, we recommend additional security measures, including changing default passwords, deactivating unnecessary services, and setting up managed database and secret storage providers with regular backups. While these aspects are beyond the scope of this book, we will provide corresponding notes with our recommendations when appropriate.

Hardware recommendations

The Hyperledger Identus cloud agent alone doesn't require too much hardware, any instance with 2GB-4GB ram will run it for testing purposes. Of course as you scale in usage you will naturally want more ram available to handle higher concurrent loads.

The Cardano wallet and DB sync components on the other hand are going to need a lot more resources due to the `mainnet` requirements, according to the official documentation, to run a full Cardano node you need:

- 200GB of disk space (for the history of blocks)
- 24GB of RAM (for the current UTxO set)

Of course, on a production environment you may want to run your agent, wallet and db-sync on different machines connected through a VPN or SSH tunnel in order to isolate them and improve security, e.g., your Cardano wallet may be only connecting to a Cardano node, but not exposed to the Internet. You may also reuse and share your Cardano node instance for your wallet and db-sync components, reducing your hardware requirements.

There are many ways to setup your infrastructure and at least in the beginning, the Cardano node is the most resource demanding of all components.

Configuration

In order to setup `preprod` we recommend copying the config files from the Identus cloud agent and making your own modifications. This is because we will need to modify the docker compose file and that is currently shared among every other type of install such as `local`, `dev` and `multi`. So, to avoid making modifications that will conflict with those defaults, we recommend copying and merging the config into it's own file.

Preparing base config files

1. Copy `infrastructure/local` config into your `preprod` destination, e.g. standing in the `cloud-agent` root directory:

```
cp -rf infrastructure/local infrastructure/preprod
```

2. Copy `infrastructure/shared/docker-compose.yml` for `preprod`:

```
cp infrastructure/shared/docker-compose.yml infrastructure/shared/docker-compose-preprod.yml
```

3. Modify `infrastructure/preprod/run.sh` to point to the new docker compose, the `diff` between the files should look like this:

```diff
--- local/run.sh    2024-06-18 13:08:47
+++ preprod/run.sh  2024-09-16 17:03:56
@@ -125,5 +125,5 @@

 PORT=${PORT} NETWORK=${NETWORK} DOCKERHOST=${DOCKERHOST} docker compose \
    -p ${NAME} \
-   -f ${SCRIPT_DIR}/../shared/docker-compose.yml \
+   -f ${SCRIPT_DIR}/../shared/docker-compose-preprod.yml \
    --env-file ${ENV_FILE} ${DEBUG} up ${BACKGROUND} ${WAIT}
```

4. Modify `docker-compse-prepod.yml` to disable `postgres` port mapping and to be able to set `DEV_MODE` trough an environment variable:

```diff
--- docker-compose.yml  2024-06-18 13:08:47
+++ docker-compose-preprod.yml  2024-09-16 17:41:36
@@ -15,8 +15,8 @@
       - pg_data_db:/var/lib/postgresql/data
       - ./postgres/init-script.sh:/docker-entrypoint-initdb.d/init-script.sh
       - ./postgres/max_conns.sql:/docker-entrypoint-initdb.d/max_conns.sql
-    ports:
-      - "127.0.0.1:${PG_PORT:-5432}:5432"
+    #ports:
+    #  - "127.0.0.1:${PG_PORT:-5432}:5432"
     healthcheck:
       test: ["CMD", "pg_isready", "-U", "postgres", "-d", "agent"]
       interval: 10s
@@ -96,7 +96,7 @@
       VAULT_ADDR: ${VAULT_ADDR:-http://vault-server:8200}
       VAULT_TOKEN: ${VAULT_DEV_ROOT_TOKEN_ID:-root}
       SECRET_STORAGE_BACKEND: postgres
```

```
-       DEV_MODE: true
+       DEV_MODE: ${DEV_MODE:-true}
        DEFAULT_WALLET_ENABLED:
        DEFAULT_WALLET_SEED:
        DEFAULT_WALLET_WEBHOOK_URL:
```

5. Modify `.env` file and add environmental variables for Cardano, please note that `NETWORK` variable conflicts with the prism node network, so we are renaming it to `NODE_CARDANO_NETWORK`, your `.env` should look like this:

```
### IDENTUS
AGENT_VERSION=1.33.0
PRISM_NODE_VERSION=2.2.1
VAULT_DEV_ROOT_TOKEN_ID=root

### CARDANO
NODE_CARDANO_NETWORK=preprod
NODE_DB=$PWD/cardano/node-db
WALLET_DB=$PWD/cardano/wallet-db
NODE_CONFIGS=$PWD/cardano/configs
NODE_SOCKET_NAME=node.socket
NODE_SOCKET_DIR=$PWD/cardano/ipc
NODE_TAG=9.1.1
WALLET_TAG=2024.9.3
WALLET_PORT=8090
WALLET_UI_PORT=8091
```

6. Update your `docker-composer-preprod.yml` to add the Cardano wallet service.

```yaml
---
version: "3.8"

services:
  ##########################
  # Database
  ##########################
  db:
    image: postgres:13
    environment:
      POSTGRES_MULTIPLE_DATABASES: "pollux,connect,agent,node_db"
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    volumes:
      - pg_data_db:/var/lib/postgresql/data
      - ./postgres/init-script.sh:/docker-entrypoint-initdb.d/init-script.sh
      - ./postgres/max_conns.sql:/docker-entrypoint-initdb.d/max_conns.sql
    #ports:
    #  - "127.0.0.1:${PG_PORT:-5432}:5432"
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "postgres", "-d", "agent"]
      interval: 10s
      timeout: 5s
      retries: 5

  pgadmin:
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL:-pgadmin4@pgadmin.org}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD:-admin}
      PGADMIN_CONFIG_SERVER_MODE: "False"
    volumes:
      - pgadmin:/var/lib/pgadmin
```

```yaml
  ports:
    - "127.0.0.1:${PGADMIN_PORT:-5050}:80"
  depends_on:
    db:
      condition: service_healthy
  profiles:
    - debug

##########################
# Services
##########################

prism-node:
  image: ghcr.io/input-output-hk/prism-node:${PRISM_NODE_VERSION}
  environment:
    NODE_PSQL_HOST: db:5432
    NODE_REFRESH_AND_SUBMIT_PERIOD:
    NODE_MOVE_SCHEDULED_TO_PENDING_PERIOD:
    NODE_WALLET_MAX_TPS:
  depends_on:
    db:
      condition: service_healthy

vault-server:
  image: hashicorp/vault:latest
  #    ports:
  #      - "8200:8200"
  environment:
    VAULT_ADDR: "http://0.0.0.0:8200"
    VAULT_DEV_ROOT_TOKEN_ID: ${VAULT_DEV_ROOT_TOKEN_ID}
  command: server -dev -dev-root-token-id=${VAULT_DEV_ROOT_TOKEN_ID}
  cap_add:
    - IPC_LOCK
```

```yaml
    healthcheck:
      test: ["CMD", "vault", "status"]
      interval: 10s
      timeout: 5s
      retries: 5

  cloud-agent:
    image: ghcr.io/hyperledger/identus-cloud-agent:${AGENT_VERSION}
    environment:
      POLLUX_DB_HOST: db
      POLLUX_DB_PORT: 5432
      POLLUX_DB_NAME: pollux
      POLLUX_DB_USER: postgres
      POLLUX_DB_PASSWORD: postgres
      CONNECT_DB_HOST: db
      CONNECT_DB_PORT: 5432
      CONNECT_DB_NAME: connect
      CONNECT_DB_USER: postgres
      CONNECT_DB_PASSWORD: postgres
      AGENT_DB_HOST: db
      AGENT_DB_PORT: 5432
      AGENT_DB_NAME: agent
      AGENT_DB_USER: postgres
      AGENT_DB_PASSWORD: postgres
      POLLUX_STATUS_LIST_REGISTRY_PUBLIC_URL: http://${DOCKERHOST}:${PORT}/cl
      DIDCOMM_SERVICE_URL: http://${DOCKERHOST}:${PORT}/didcomm
      REST_SERVICE_URL: http://${DOCKERHOST}:${PORT}/cloud-agent
      PRISM_NODE_HOST: prism-node
      PRISM_NODE_PORT: 50053
      VAULT_ADDR: ${VAULT_ADDR:-http://vault-server:8200}
      VAULT_TOKEN: ${VAULT_DEV_ROOT_TOKEN_ID:-root}
      SECRET_STORAGE_BACKEND: postgres
      DEV_MODE: ${DEV_MODE:-true}
```

```
      DEFAULT_WALLET_ENABLED:
      DEFAULT_WALLET_SEED:
      DEFAULT_WALLET_WEBHOOK_URL:
      DEFAULT_WALLET_WEBHOOK_API_KEY:
      DEFAULT_WALLET_AUTH_API_KEY:
      GLOBAL_WEBHOOK_URL:
      GLOBAL_WEBHOOK_API_KEY:
      WEBHOOK_PARALLELISM:
      ADMIN_TOKEN:
      API_KEY_SALT:
      API_KEY_ENABLED:
      API_KEY_AUTHENTICATE_AS_DEFAULT_USER:
      API_KEY_AUTO_PROVISIONING:
    depends_on:
      db:
        condition: service_healthy
      prism-node:
        condition: service_started
      vault-server:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://cloud-agent:8085/_system/health"]
      interval: 30s
      timeout: 10s
      retries: 5
    extra_hosts:
      - "host.docker.internal:host-gateway"

  swagger-ui:
    image: swaggerapi/swagger-ui:v5.1.0
    environment:
      - 'URLS=[
        { name: "Cloud Agent", url: "/docs/cloud-agent/api/docs.yaml" }
```

```
      ]'

  apisix:
    image: apache/apisix:2.15.0-alpine
    volumes:
      - ./apisix/conf/apisix.yaml:/usr/local/apisix/conf/apisix.yaml:ro
      - ./apisix/conf/config.yaml:/usr/local/apisix/conf/config.yaml:ro
    ports:
      - "${PORT}:9080/tcp"
    depends_on:
      - cloud-agent
      - swagger-ui

  ##########################
  # Cardano
  ##########################

  cardano-node:
    image: cardanofoundation/cardano-wallet:${WALLET_TAG}
    environment:
      CARDANO_NODE_SOCKET_PATH: /ipc/${NODE_SOCKET_NAME}
    volumes:
      - ${NODE_DB}:/data
      - ${NODE_SOCKET_DIR}:/ipc
      - ${NODE_CONFIGS}:/configs
    restart: on-failure
    #user: ${USER_ID}:${GROUP_ID}
    logging:
      driver: "json-file"
      options:
        compress: "true"
        max-file: "10"
        max-size: "50m"
```

```yaml
    entrypoint: []
    command: >
      cardano-node run --topology /configs/cardano/${NODE_CARDANO_NETWORK}/topology.json
        --database-path /data
        --socket-path /ipc/node.socket
        --config /configs/cardano/${NODE_CARDANO_NETWORK}/config.json
        +RTS -N -A16m -qg -qb -RTS

  cardano-wallet:
    image: cardanofoundation/cardano-wallet:${WALLET_TAG}
    volumes:
      - ${WALLET_DB}:/wallet-db
      - ${NODE_SOCKET_DIR}:/ipc
      - ${NODE_CONFIGS}:/configs
    ports:
      - 127.0.0.1:${WALLET_PORT}:8090
      - 127.0.0.1:${WALLET_UI_PORT}:8091
    environment:
      NETWORK: ${NODE_CARDANO_NETWORK}
    entrypoint: []
    command: >
      cardano-wallet serve
        --node-socket /ipc/${NODE_SOCKET_NAME}
        --database /wallet-db
        --listen-address 0.0.0.0
        --testnet /configs/cardano/${NODE_CARDANO_NETWORK}/byron-genesis.json

    #user: ${USER_ID}:${GROUP_ID}
    restart: on-failure
    logging:
      driver: "json-file"
      options:
        compress: "true"
```

```
        max-file: "10"
        max-size: "50m"
  icarus:
    image: piotrstachyra/icarus:v2023-04-14
    ports:
      - 127.0.0.1:4444:4444
    network_mode: "host"
    restart: on-failure

volumes:
  pg_data_db:
  pgadmin:
  node-ipc:
# Temporary commit network setting due to e2e CI bug
# to be enabled later after debugging
#networks:
#  default:
#    name: ${NETWORK}
```

Cardano wallet

In order to publish DIDs into the VDR, we need to setup our Cardano wallet, the following steps will incorporate the Cardano wallet docker config into our Identus docker config. We will merge whats on https://github.com/cardano-foundation/cardano-wallet/tree/master/run/preprod/docker into our cloud agent setup.

1. Create `snapshot.sh`

```
#! /usr/bin/env -S nix shell 'nixpkgs#curl' 'nixpkgs#lz4' 'nixpkgs#gnutar' --
# shellcheck shell=bash
```

```
set -euo pipefail

# shellcheck disable=SC1091
source .env

# Define a local db if NODE_DB is not set
if [[ -z "${NODE_DB-}" ]]; then
    LOCAL_NODE_DB=./databases/node-db
    mkdir -p $LOCAL_NODE_DB
    NODE_DB=$LOCAL_NODE_DB
fi

# Clean the db directory
rm -rf "${NODE_DB:?}"/*

echo "Network: $NETWORK"

case "$NETWORK" in
    preprod)
        SNAPSHOT_NAME=$(curl -s https://downloads.csnapshots.io/testnet/testnet-db-snapshot.
        echo "Snapshot name: $SNAPSHOT_NAME"
        SNAPSHOT_URL="https://downloads.csnapshots.io/testnet/$SNAPSHOT_NAME"
        ;;
    mainnet)
        SNAPSHOT_NAME=$(curl -s https://downloads.csnapshots.io/mainnet/mainnet-db-snapshot.
        echo "Snapshot name: $SNAPSHOT_NAME"
        SNAPSHOT_URL="https://downloads.csnapshots.io/mainnet/$SNAPSHOT_NAME"
        ;;
    *)
        echo "Error: Invalid network $NETWORK"
        exit 1
        ;;
esac
```

```
echo "Downloading the snapshot..."

if [ -n "${LINK_TEST:-}" ]; then
    echo "Link test enabled"
    echo "Snapshot URL: $SNAPSHOT_URL"
    curl -f -LI "$SNAPSHOT_URL" > /dev/null
    curl -r 0-1000000 -SL "$SNAPSHOT_URL" > /dev/null
    exit 0
fi

curl -SL "$SNAPSHOT_URL" | lz4 -c -d - | tar -x -C "$NODE_DB"

mv -f "$NODE_DB"/db/* "$NODE_DB"/
rm -rf "$NODE_DB"/db

echo "Snapshot downloaded and extracted to $NODE_DB"
```

2. Run `snapshot.sh` in order to sync the required ledger snapshot (you will need `curl`, `lz4` and `tar` installed in your system):

```
bash snapshot.sh
Network: preprod
Snapshot name: testnet-db-70689600.tar.lz4
Downloading the snapshot...
...
Snapshot downloaded and extracted to ./cardano/node-db
```

3. Copy Cardano wallet configs into `./cardano/configs`
4. Run `./refresh.sh` to download the configs.
5. Confirm configs directory for `preprod` look like this:

```
ls cardano/configs/cardano/preprod
alonzo-genesis.json  byron-genesis.json   config.json          conway-genesis.json  download
```

In this chapter we will discuss how to prepare a production envioronment for an Identus application.

We will discuss:

- Hardware recommendations
- Production configuration and security
- Lock down Docker / Postgres (default password hole, etc)
- SSL
- Multi Tenancy
- Keycloak
- Testnet / Preprod env
- Connecting to Mainnet
- Set up Cardano Wallet
- Key Management with HashiCorp
- Connecting to Cardano
- Running dbSync

Maintenance

Mastering Identus means maintaing your application once it's launched.

In this chapter we will cover:

- Restarting and cleanup
- Observability
  - Manage nodes / Memory
  - Performance Testing
  - Analytics with BlockTrust Analytics
- Upgrading Agents
  - How to minimize downtime
- Hashicorp
  - Key Management
  - Key rotation

Section V - Addendum

Trust Registries

Overview of the concept of a Trust Registry

- What role they play in SSI
- Trust Over IP Trust Registry Spec
- Highlight any real world examples if they exist by book completion

Continuing on Your Journey

Information about becoming a Contributor to Identus

- How to contribute to the source code or documentation

Information about how to get involved in the SSI community outside of Identus

Trust over IP (ToIP)

Trust Over IP (ToIP) comprises over 300 organizations from diverse industries collaborating to advance Decentralized Identity through ideas and software. According to ToIP, "We develop tools and specifications to help communities of any size use digital networks to build and strengthen trust between participants."

The ToIP Stack, published in 2019, describes how the four layers (DIDs, DIDComm Protocol, Data Exchange Protocols, and Application Ecosystems) form a secure, scalable, and interoperable digital trust framework. Introduction to ToIP, Version 2.0, released in 2021, is an excellent resource for anyone catching up on the Digital Identity problem and solution. Membership options include free Contributor accounts and paid memberships, providing access to ToIP's resources and Slack channels for collaboration.

To join Trust Over IP, you need to create a free Linux Foundation account and then complete your ToIP application here.

All members agree to open, non-competitive participation, so please have your legal team review all associated documents during the onboarding process.

For further details, refer to The ToIP Stack and Overview.

Decentralized Identity Foundation (DIF)

Appendices

Errata

Errata goes here

We will list any bugs that may have been “printed” in certain editions of the book.

Glossary

Glossary or Index here

This will reference key concepts and Identus terms and where they are mentioned in the book, allowing someone to look up a term and know what section it is referenced in.

TODO: First draft will just lay down the glossary terms, we want to add detailed descriptions later.

VDR Verifiable Data Registry