# Introduction

Within Hyperledger Indy/Sovrin there is the notion of an agent or an agency respectively. The different terms are described below. In our current implementation and the IdentityChain project we use the term cloud agent as an intermediary between the ledger and the mobile app. The cloud agent fulfills the purpose of forwarding messages if the app is not online or reachable.

# Motivation

Within the IdentityChain (IDC) project the IndySDK is used on the IdentityChain API layer and the Android app layer. The API abstracts the IndySDK functionality to ease the use for developers. The Android app which is written in Java is using the IndySDK Java wrapper for exchanging values and methods with the ledger. The motivation for the introduction of a cloud agent within IdentityChain was the fact that credentials will be send, updated and revoked from the issuer's interface, in our case a web application using the IdentityChain-API. In a scenario where the app is not online and/or connected with the ledger, the IdentityChain Cloud Agent (IDC-CA) is handling temporal storing and forwarding of messages, i.e., verifiable credentials, using the Google Firebase services and APIs. The IDC-CA is also used to initiate a setup of values for the IDC Mobile App. The IDC-CA can be seen as a unique endpoint for the mobile app. In future there will be additional functionalities and use cases extending the IDC-CA. This is also inline with the notion of agents within Hyperledger Indy itself, e.g., surrogate functions, such as automated replies to certain proof requests.

# The different notions of Agents

In the following sub-sections the notion of the IDC agents is described. It may differ from the Hyperledger Indy/Sovrin meaning, which ca be found under:

https://github.com/hyperledger/indy-agent/tree/master/docs

## Edge Agent

An Edge Agent is positioned close to the user interface. If the hardware and network of the edge device is capable enough, i.e., a PC or laptop, the edge agent can talk to the ledger directly and is also hosting a wallet which includes all keys and credentials. This can also be a more potent machine, i.e., a server in an institution, such as a bank, which provides functionalities for user interfaces, such as web-based once. This edge agent is then called institutional edge agent.

Otherwise, there are mobile edge agents for devices which are not always online and need additional functionalities to connect to other agents. In our case this is a mobile app, which is the interface for the identity holder and is connected to a cloud agent.

#### Mobile Edge Agent

As described above, a mobile edge agent is a user interface device, such as a smartphone which is not always online and is connected to a cloud agent which serves it as a message broker, etc. In our case the IDChain mobile Android app is an mobile edge agent.

### Institutional Edge Agent

An institutional edge agent is capable of communicating with other edge and cloud agents and serves as the interface for the employees of the institution. In our case the institutional edge agent is the IndySDK together with the REST-API serving the front-end Admin PortalUI.

### Cloud Agent

A cloud agent is serving devices which have less capabilities such as smartphone apps. It redirects messages without knowing their contents to the specific edge agent. In our case we use key-value stores to be able to forward the incoming messages to the correct mobile using Firebase service from Google. The cloud agent can either run at an agency (service), see below or on the controlled hardware of the identity holder, such as a home NAS, Router, or similar.n It is to be noted that a cloud agent serving one app should have different endpoints for the different pairwise connections as it ensures more privacy for the identity holder. As of now, we neglect this in the IDChain project implementation, as the current version of the Indy ledger does not support complex endpoints to be stored on the ledger, at the moment only IP and Port can be stored. Agency

An agency in our understanding is the hoster of cloud agents for identity holders, this can be a telephone operator, an IT service provider, bank or another trusted entity.

### Hub (text from github indy-agent)

Additionally there is the notion of a hub in Indy/Sovrin

A Hub is much like a Cloud Agent, but rather than focusing only on messaging (transport) as defined above for a Cloud Agent, the Hub also stores and shares

data (potentially including Verifiable Credentials), on behalf of its owner. All of the data held by the Hub is en/decrypted by the Edge Agent, so it is the data that moves between the Edge and Hub, and not keys. The Hub storage can (kind of) be thought of as a remote version of a Wallet without the keys, but is intended to hold more than just the Verifiable Credentials of an Edge Agent wallet. The idea is that the user can push lots of, for example, app-related data to the Hub, and a Service would be granted permission by the Owner to directly access the data without having to go to the Edge Agent. For example, a Hub-centric music service would store the owner's config information and playlists on the Hub, and the Service would fetch the data from the Hub on use instead of storing it on it's own servers. # Overview The following figure illustrates the overall IDC components also linked to the IndySDK and the ledger. At the top there are the three user interfaces for the issuer, the verifier, and an IDC-AdminUI. They are all accessing the IDC-API, which abstracts the IndySDK for the components. The IDC-Schema-Compiler which can be accessed via the IDC-AdminUI is also integrated in the API. The IDC-CA is an intermediary between the IDC-Mobile-App and the IDC-UIs and API. The IDC-Mobile-App is additionally using the IndySDK Java wrapper to access the ledger directly.

### Software Components

## Cloud Agent Message Flow

This section illustrates the message flow of the Cloud Agent and the corresponding actors.

#### Prerequisites

- It is assumed that the TA has been previously onboarded and has Trust Anchor privileges to write on the ledger.
- FCM limits the message payload to 4KB. Messages with payloads >=4KB will be temporarily stored on the CA. The CA provides a unique URL to retrieve the message payload.
- Follow the IDC_CA_API documentation and the section above for specific formats on exchanged messages.

#### Sequence Flow

1. The IDC-CA requires an initial onboarding from a Trust Anchor (TA) in order to be known on the Ledger. Therefore it sends a connection request to the TA, which should be acknowledged by a connection response. The

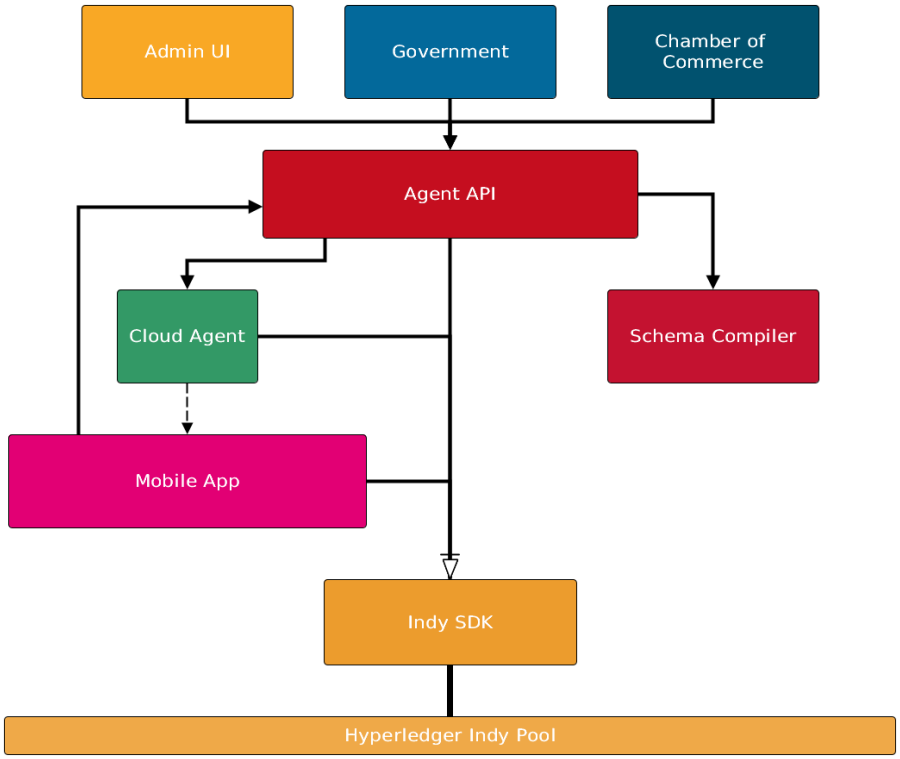


Figure 1: IDC components

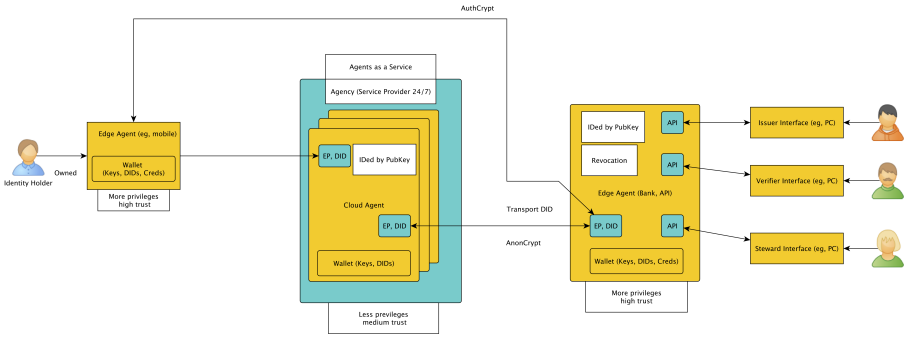


Figure 2: Agent Role Actors

TA will onboard the CA with a NYM-request, including the CA_DID, the CA_VerificationKey, and the CA_Endpoint, to the ledger.

2. The IDC-Mobile-App sends a connection request to a known CA containing the App_DID, the App_Verification-Key and the Endpoint in form of a Firebase Token. The CA replies a connection response containing the CA_DID and CA_Endpoint which can be used by the APP for further communication.

3. The APP follows the pairwise connection pattern with responding to a TA_Connection_Offer with a Connection_Request and provides the CA_DID, and CA_Endpoint to be reached by the TA alongside its own DID and verification key (App_DID, App_VerKey). The TA sends a NYM_request to the ledger including the TA_APP_DID.

4. The TA sends authcrypted payload messages regarding the APP to the provided IDC-CA_Endpoint anoncrypted with the key of the CA. The CA decrypts the anoncrypted message with it's private key and forwards the payload message anoncrypted to the APP through the provided Firebase Token with FCM. The APP can directly respond to the TA_Endpoint with an authcrypted message.

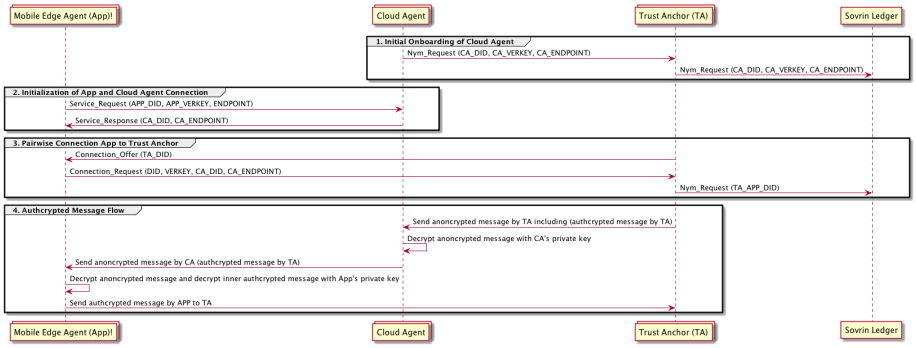


Figure 3: CA Message Flow

# Working together with other IDC components

## Cloud Agent Message Stack

The IDC-CA message stack is as follows. Messages from the IDC-App to the CA are transmitted via https. Messages from the CA to the app are transmitted with Google Firebase messages, unless the message is bigger than XX, in that case the Firebase message includes a link to download the message directly to the

app. Messages to the CA from any other application or the API's perspective are done via https. The figure below illustrates the message stack of the CA.

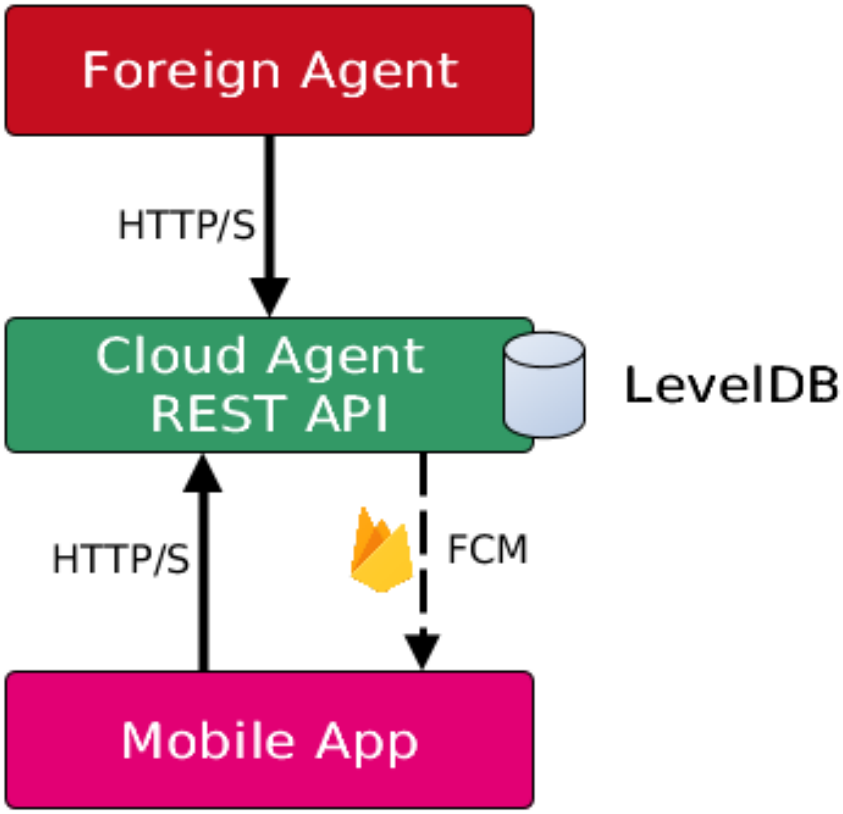


Figure 4: IDC-CA Message Stack

### Differences between the Indy/Sovrin CA and IDC CA

## Message Formats

In Indy/Sovrin message flow and message structures/formats are vital for the different roles and software components. The following gives you an overview of the encryption details and the message formats in detail. There are JSON-based and used in different contexts.

### Encryption

Messages exchanged between the CA and the mobile app and a foreign agents are not only secured via possible https connections but also encrypted in two different ways:

- `anoncryped`
- `authcrypted`

The explanation below is focusing on this difference and motives why it is used.

#### anoncrypted

The anonymous-encryption schema is designed for the sending of messages to a recipient which has been given its public key. Only the recipient can decrypt these messages, using its private key. A anoncrypted message is encrypted for the receiver by the sender. The receiver does not know the sender and therefore cannot authenticate her/him/it. The sender send an anonymous message, e.g., at the beginning of a new onboarding process. While the receiver can verify the integrity of the message, it cannot verify the identity of the sender.

#### authcrypted

Authenticated encryption (authcrypt) is designed for sending of a confidential message specifically for the Recipient. The sender can compute a shared secret key using the recipient's public key (verkey) and his secret (signing) key. The recipient can compute exactly the same shared secret key using the sender's public key (verkey) and his secret (signing) key. That shared secret key can be used to verify that the encrypted message was not tampered with, before eventually decrypting it. A authcrypted message is encrypted for the receiver by the sender. The receiver knows the sender from a previous, e.g., onboarding process and therefore can at this point authenticate her/him/it. The receiver can verify the integrity of the message and the identity of the sender.

## Agent-to-Agent Communication

Messages sent between agents have the following format:

```
{
  message: <message>
}
```

With `<message>` being one of the message formats described below, mostly as an anoncrypted string (+ authcrypted inner message, if applicable).

# Message Formats

Heavily influenced by: https://github.com/hyperledger/indy-agent/tree/master/docs

Outer messages are always anon-crypted and base64-encoded for the endpointDid (except for connection offer). Inner messages are always auth-crypted and base64-encoded after pairwise is established (from connection acknowledgement onwards).

NOTE: Message types have been changed on indy-agent side, we are still using the initial ones.

1. Connection Offer
2. Connection Request
3. Connection Response
4. Connection Acknowledgement
5. Credential Offer
6. Credential Request
7. Credential
8. Proof Request
9. Proof

## Connection Offer

- renamed offer_nonce to nonce (as it is also implemented in indy-agent, docs differ here)
- added `message.data` field
- message is not encrypted

```
{
    // connection offer nonce
    // necessary when implementing an agency as an id to route to indended agent/wallet
    id: <offer_nonce>,
    type: "urn:sovrin:agent:message_type:sovrin.org/connection_offer",
    message: {
        // Endpoint DID also found in ledger. Together with attached endpoint
        did: <did>,
        // (optional) Required if Endpoint DID not stored on Ledger
        verkey: <verkey>,
        // (optional) Required if Endpoint DID not stored on Ledger or has no diddoc
        endpoint: <endpoint>,
        // connection offer nonce
        nonce: <offer_nonce>,
        // additional field: may contain additional related data such as description, logo,
        data: {}
```

```
    }
}
```

### Connection Request

- added verkey for cases where the did is not on the ledger (e.g. onboarding or pairwise connection without writing the did on the ledger), similar to how there is already a verkey in connection response
- rename `endpoint_did` to `endpointDid` in line with indy-agent implementation
- outer message/payload is anoncrypted for recipient endpoint did
- inner message is not encrypted

```
{
    // if there was a previous offer, then this is the offer nonce, else request nonce
    id: <offer_nonce / request_nonce>,
    type: "urn:sovrin:agent:message_type:sovrin.org/connection_request",
    // inner message is not encrypted
    message: {
        did: <myNewDid>,
        // additional field: (optional) required if did not stored on Ledger
        verkey: <myVerkey>,
        // (optional) required if endpoint DID is not stored on the Ledger or has no diddoc
        endpointDid: <endpointDid>,
        // (optional) required if endpoint DID is not stored on the Ledger or does not conta
        endpoint: <endpoint>,
        nonce: <request_nonce>
    }
}
```

### Connection Response

- rename `request_nonce` to just `nonce` in message for consistency (as it is also done in indy-agent implementation)
- add `aud` field (as in indy-agent implementation)
- outer message/payload is anoncrypted for recipient endpoint did
- inner message now additionally anoncrypted for recipient pairwise did

```
{
    id: <request_nonce>,
    // additional field: recipient pairwise did
    aud: <theirDid>,
    type: "urn:sovrin:agent:message_type:sovrin.org/connection_response",
    // inner message is anoncrypted and base64 encoded
    message: {
```

```
        // my pairwise did and verkey
        did: <myDid>,
        verkey: <myVerkey>,
        nonce: <requestNonce>
    }
}
```

## Connection Acknowledgement

- outer message/payload is anoncrypted for recipient endpoint did
- inner message is now authcrypted using both pairwise dids

```
connection_acknowledgement: {
    // pairwise did of sender
    id: <myDid>,
    type: "urn:sovrin:agent:message_type:sovrin.org/connection_acknowledge",
    // inner message is authcrypted and base64 encoded
    message: "SUCCESS"
}
```

## Credential Offer

- As of 2018-10-11, this message is not defined by indy-agent
- inner message is authcrypted

```
{ // anoncrypted
    id: <offerNonce>,
    // pairwise did of the sender
    origin: <myDid>,
    type: "urn:sovrin:agent:message_type:sovrin.org/credential_offer",
    message: { // authcrypted
        "schema_id": string,
        "cred_def_id": string,
        // Fields below can depend on Cred Def type
        "nonce": <offerNonce>,
        "key_correctness_proof" : <key_correctness_proof>
    } // authcrypted
} // anoncrypted
```

## Credential Request

- As of 2018-10-11, this message is not defined by indy-agent
- inner message is authcrypted

```
{ // anoncrypted
    id: <offerNonce>,
    // pairwise did of sender
    origin: <myDid>,
    type: "urn:sovrin:agent:message_type:sovrin.org/credential_request",
    message: { // authcrypted
        "prover_did" : string,
        "cred_def_id" : string,
        // Fields below can depend on Cred Def type
        "blinded_ms" : <blinded_master_secret>,
        "blinded_ms_correctness_proof" : <blinded_ms_correctness_proof>,
        "nonce": <request_nonce>
    } // authcrypted
} // anoncrypted
```

## Credential

- As of 2018-10-11, this message is not defined by indy-agent
- inner message is authcrypted

```
{ // anoncrypted
    id: <request_nonce>,
    // pairwise did of the sender
    origin: <myDid>,
    type: "urn:sovrin:agent:message_type:sovrin.org/credential",
    message: { // authcrypted
        "schema_id": string,
        "cred_def_id": string,
        "rev_reg_def_id", Optional<string>,
        "values": <see credValues above>,
        // Fields below can depend on Cred Def type
        "signature": <signature>,
        "signature_correctness_proof": <signature_correctness_proof>
    } // authcrypted
} // anoncrypted
```

## Proof Request

- As of 2018-10-11, this message is not defined by indy-agent
- inner message is authcrypted
- 2018-11-01: nonce MUST be numerical

```
{ // anoncrypted
    "id": "<requestNonce>",
    // pairwise did of sender
```

```
    "origin": "<myDid>",
    "type": "urn:sovrin:agent:message_type:sovrin.org/proof_request",
    "message": { // authcrypted
        // proof request name
        "name": "<name>",
        // proof request version
        "version": "<version>",
        // proof request nonce
        "nonce": "<requestNonce>"
        "requested_attributes": {
            "attr1_referent": {
                // attribute name, e.g. 'firstname'
                "name": "<attr_name>",
                // restrictions on where the attribute comes from
                // no restrictions mean it may be self-attested
                "restrictions": [{ "cred_def_id": "<cred_def_id>" }]
            }
            "attr2_referent": {
                // self-attested attribtue
                "name": "<attr_name>"
            }
        },
        "requested_predicates": {
            "predicate1_referent": {
                // predicate name
                "name": "<predicate_name>",
                // predicate type, as of 2018-10-24: only '>=' seems to be supported
                "p_type": ">=",
                // predicate value to check against
                "p_value": "<value>",
                // restrictions similar to requested_attributes
                "restrictions": [{ "cred_def_id": "<cred_def_id>" }]
            }
        }
    } // authcrypted
} // anoncrypted
```

## Proof

- As of 2018-10-11, this message is not defined by indy-agent
- inner message is authcrypted
- The below proof is an example taken from https://www.npmjs.com/package/indy-sdk#provercreateproof–wh-proofreq-requestedcredentials-mastersecretname-schemas-credentialdefs-revstates—-proof (2018-10-25)
- 2018-11-01: nonce MUST be numerical

```
{ // anoncrypted
    id: <requestNonce>,
    // pairwise did of sender
    origin: <myDid>,
    type: "urn:sovrin:agent:message_type:sovrin.org/proof",
    message: { // authcrypted, below is just an example, this is whatever indy-sdk returns
        "requested_proof": {
            "revealed_attrs": {
                "requested_attr1_id": {sub_proof_index: number, raw: string, encoded: string
                "requested_attr4_id": {sub_proof_index: number: string, encoded: string},
            },
            "unrevealed_attrs": {
                "requested_attr3_id": {sub_proof_index: number}
            },
            "self_attested_attrs": {
                "requested_attr2_id": self_attested_value,
            },
            "requested_predicates": {
                "requested_predicate_1_referent": {sub_proof_index: int},
                "requested_predicate_2_referent": {sub_proof_index: int},
            }
        },
        "proof": {
            "proofs": [ <credential_proof>, <credential_proof>, <credential_proof> ],
            "aggregated_proof": <aggregated_proof>
        },
        "identifiers": [{schema_id, cred_def_id, Optional<rev_reg_id>, Optional<timestamp>}]
    } // authcrypted
} // anoncrypted
```

## Setup

## IdentityChain Commons

This repository holds commonly used components, configurations, and tools for the IdentityChain project.

## Guidelines

Add new stuff by either

- creating a directory for it or

- create a new branch normally
- create a new branch with no ancestors from tag empty:

```
git checkout --orphan deployment/aws empty
```

Update the readme, add your changes, submodules, git subtrees, commit, and push the new branch to origin as usual.

# Building, starting and stopping services

## Building IDC_CA service

Building of the services requires all environment variables used in containers and denoted in composition file to be set in one central .env file, see *Environment variables used in API project* section. Note that building services with environment variables not set may look succeeded, however applications in the containers will likely fail.

To build services use the command:

```
./build.sh
```

## Staring services

To start services using default docker-compose.yaml:

```
./up.sh
```

To start services using another compose script, e.g. *special.yaml*

```
./up.sh special
```

## Deployment of the Cloud Agent/Agency* API

The deployment of the cloud agent/agency is described in the following.

## Prerequisites

For a brief overview on which parameters are required for a working deployment, check the example.env in the repository.

### Firebase Cloud Messaging

The implementation of the IDChain Cloud Agent implements Firebase Cloud Messaging (FCM) for communication with mobile applications. A prior registration of a project at http://console.firebase.google.com is required to use FCM. Set the following env-variables accordingly. The file we currently used was uploaded to the servers. Current name: 'eit-idchain-app-firebase-adminsdk.json'. It was generated with Bersant Google account and used for the time being.

```
FIREBASE_ADMIN_PATH='/path/to/xxx-adminsdk.json'
```

```
FIREBASE_PROJECT_URL='https://xxx.firebaseio.com'
```

### LevelDB

The Cloud Agent uses LevelDB for storing data and keeping track of the connections between unique URLs provided to App instances and FCM Tokens. Make sure the application has write access to the path set as described below.

```
DB_PATH='./data'
```

### Hyperledger Indy Pool Access

The Cloud Agent requires access to a Hyperledger Indy ledger pool in order to verify DIDs and Endpoints. For this connection a name and the pool transactions genesis of the pool is required.

```
POOL_IP=172.16.0.100
```

```
POOL_NAME=poolApi
```

```
GENESIS_TXN=pool_transactions_genesis.docker-compose
```

## Deployment Parameters

The application runs on the host and port defined in the environment variables.

```
CA_APP_HOST=127.0.1.1
```

```
CA_APP_PORT=8080
```

In addition for providing unique URLs to mobile apps, the application requires additional settings for the domain and port where the Cloud Agent is actively run at.

```
DOMAIN_HOST=127.0.1.1
```

```
DOMAIN_PORT=8080
```

Finally, the Cloud Agent implements a Hyperledger Indy Wallet which is used for storing own and foreign DID information, encrypting and decrypting outgoing and incoming messages. Therefore the following env-variables need to be set.

```
CA_WALLET_NAME='example_ca_wallet_name'
```

```
SECRET='your-secret'
```

```
CA_DID='Cloud Agent DID for initial onboarding'
```

## Deploy the App

It is recommended to deploy the Cloud Agent API behind an Nginx webserver and let Nginx act as a proxy for incoming queries to the API. Also, in order to restart and reboot easily and accessing monitoring of the API we recommend the deployment with pm2.

### PM2

Here are some helpful commands with pm2.

#### Generating Server-specific Startup Scripts

Generating server-specific startup scripts for reboot ready startup. This command will check the underlying server environment and will propose the fitting script to be executed.

```
pm2 startup
```

```
pm2 startup ubuntu
```

#### Start/Restart an application with pm2

```
pm2 start app.js -n your_app_name
```

```
pm2 restart your_app_name
```

#### List all applications

```
pm2 list
```

#### Monitor all applications

```
pm2 monit
```

#### Show application-specific information

```
pm2 show your_app_name
```

#### Show application-specific logs

```
pm2 logs your_app_name
```

### Docker

Dockerfiles added

Deployment with docker-compose thru IDChain commons, uses local Dockerfile

In IDChain commons there is a overall en file for settings, also including all other IDChain components

Note: *Cloud Agent/Agency are currently terms used interchangeably. There seems no clear and valid written description on roles and components in Hyperledger Indy yet (as of 09/03/2018).

```
How is the CA setup and configured
    Working together with ledger
    How does the ledger need to be configured
Setup of IDC App
    IDC Test App integration
Test - how to test a certain CA setup
```

## API

```
What does the CA API calls do
How can a developer use the API
```

### REST-API Structure, Routes, Methods

#### messages

#### GET /message/{id}

Retrieve a message

id* A unique string value identifying this message. This request is usually sent to clients through FCM.

### POST /messages

Send a arbitrary message to known client using FCM. Firebase token should be known to sender

#### services

### POST /services

Send a service request

#### indy-inbox

### POST /indy/{id}

Send a message to a mobile client using the unique URL endpoint provided by mobile edge agent.

id* A unique id describing the endpoint usually provided by the mobile edge agent to this cloud agent.

## Models

#### services_post

{ endpoint_did* string verkey* string endpoint* string Contains Firebase Token to be created or updated at Cloud Agent side }

#### services_response

{ endpoint_did string The endpoint did of this cloud agent endpoint string The endpoint url provided for this service request pool_config {...} }

#### messages_post

{ firebase_token* string message* string }

#### indy_inbox_post

{ message string Anoncrypted Message from other Edge Agents to be forwarded to mobile client }

# Developer

What does a dev need to know about the CA
How can the functionality be extended

# Example

Example Setup

# Tests

## Test Setup

### Test Reachability and Ports

### Test Connection to ledger

## Test API

### Test Variables

### Test Service Requests

# Code and Folder Structure

## Folder Structure

## Code Structure

# Open Issues

What needs to be done
Roadshow - Next implementation steps

# License

What is the open source license of the CA

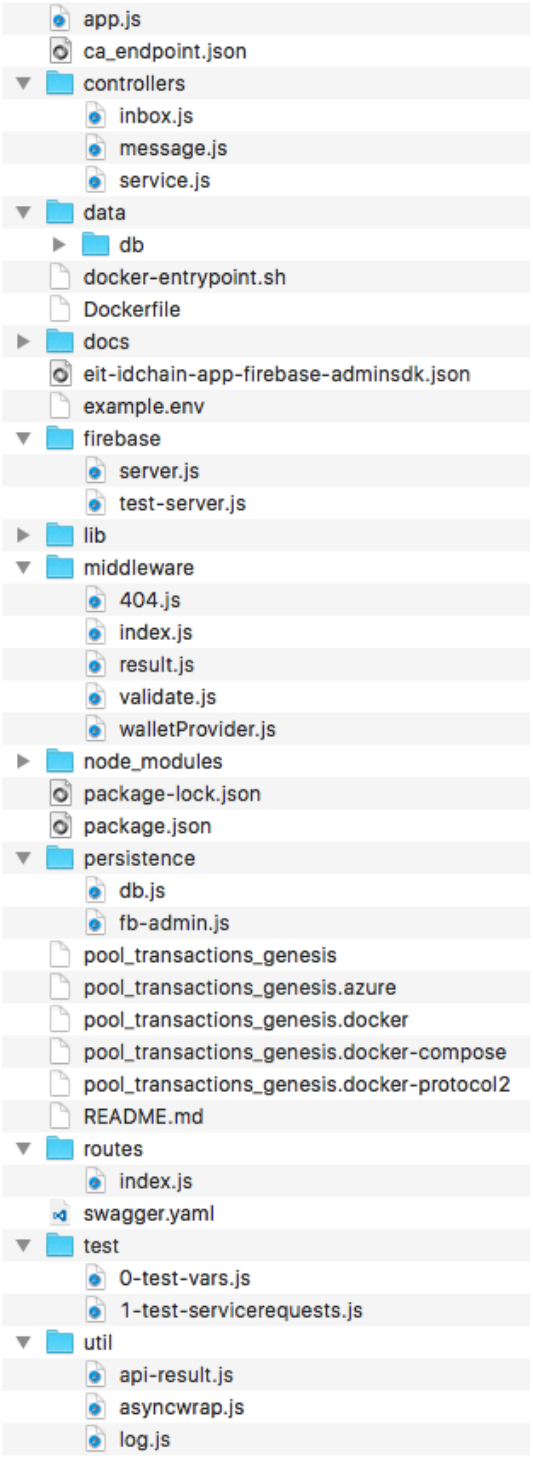


Figure 5: IDC_CA Folder Structure

# Contact

## Contact the developers

Bersant Deva
Ömer Ilhan