Tutorial On

HYPERLEDGER
INDY

Decentralized Identity Management on Blockchains

Theory and Applications of Blockchain
(CS61065)

Hyperledger Indy

Hyperledger Indy provides

- Tools
- Libraries
- reusable components

for **providing digital identities** rooted on blockchains so that they are interoperable across administrative domains, applications, and any other silo.

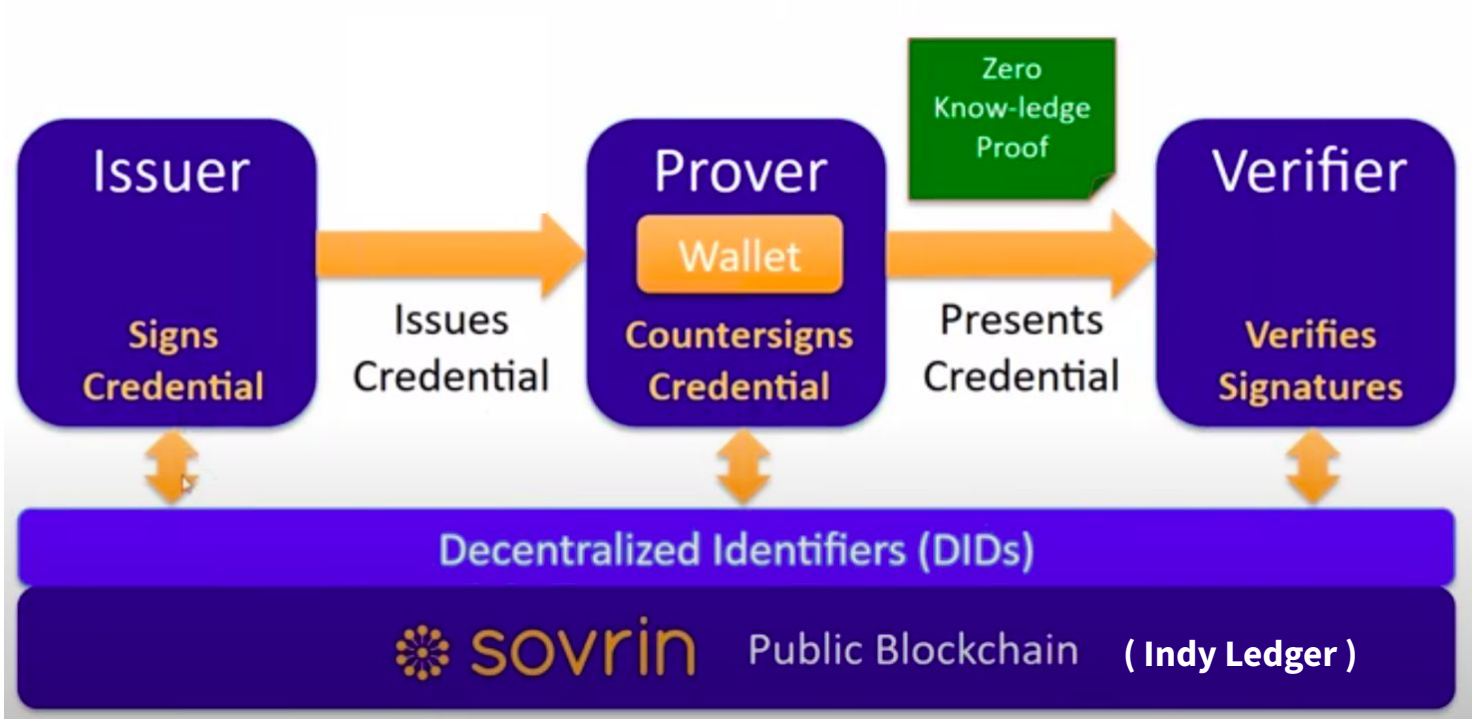
https://wiki.hyperledger.org/display/indy

Indy Key Characteristics

- Distributed ledger purpose-built for decentralized identity
- BFT by design
- **DIDs (Decentralized Identifiers)** that are globally unique and resolvable (via a ledger) without requiring any centralized resolution authority
- **Verifiable Credentials** in an interoperable format for exchange of digital identity attributes and relationships, currently in the standardization pipeline at the W3C
- **Zero Knowledge Proofs** which prove that some or all of the data in a set of Claims is true without revealing any additional information, including the identity of the Prover

https://wiki.hyperledger.org/display/indy

Indy Overview

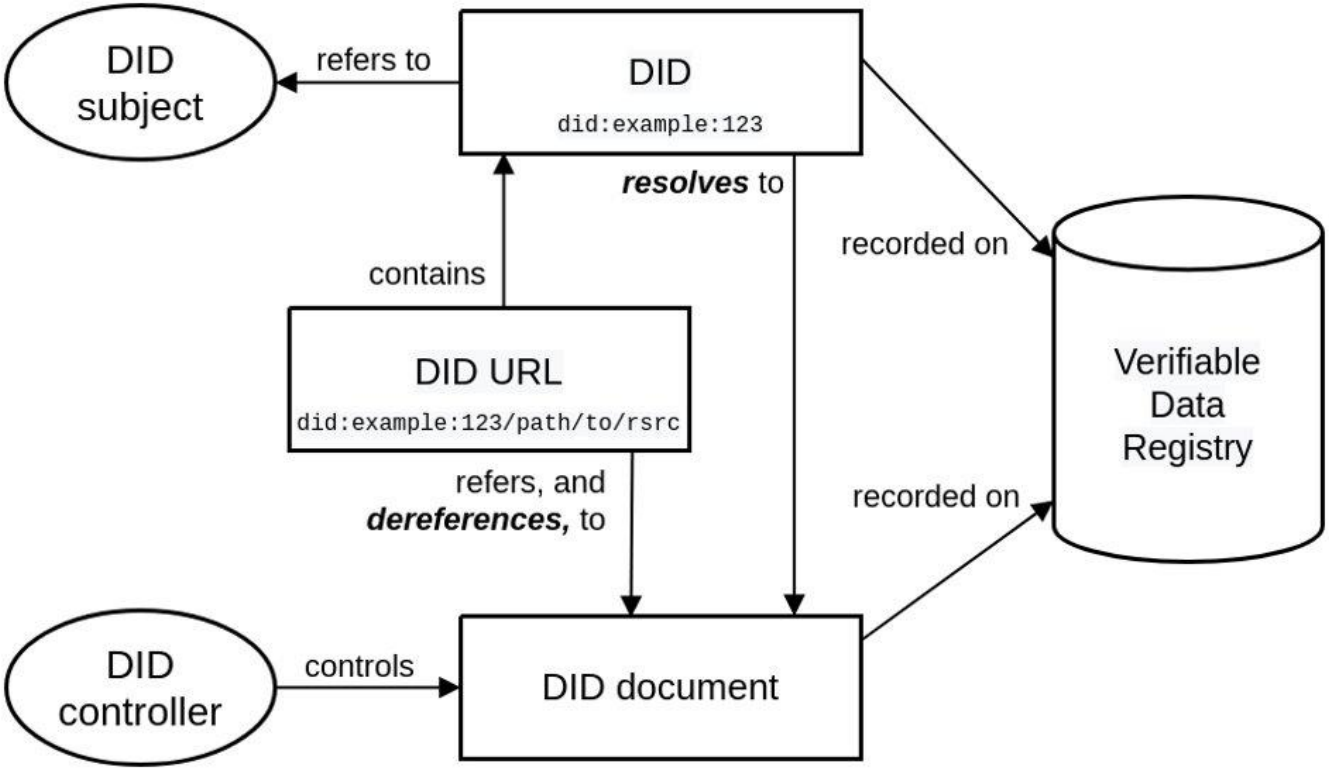


Indy Projects

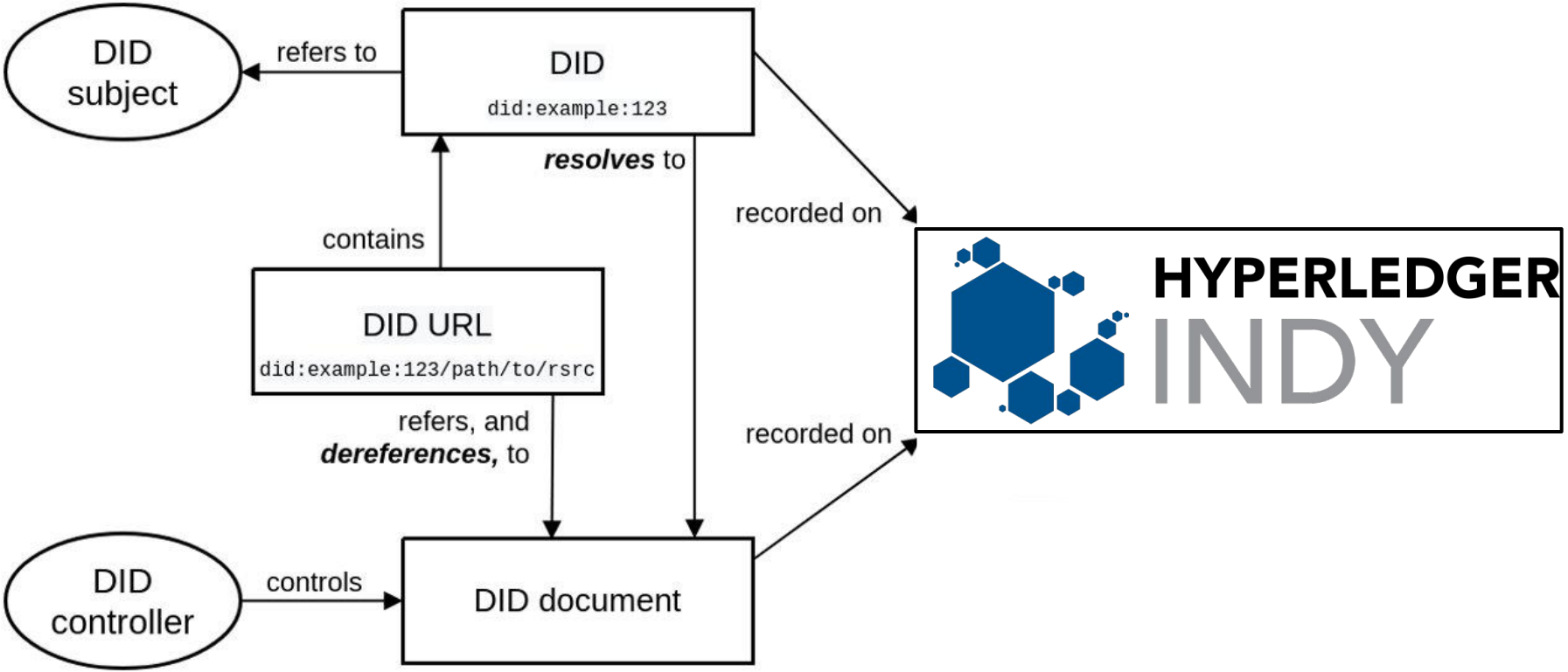
- **Indy-Plenum:**
  - Implements Byzantine Fault Tolerant Protocol
  - Used for consensus in Indy
  - Based on RBFT
  - https://github.com/Hyperledger/indy-plenum
- **Indy-Node:**
  - Implements the blockchain with Indy-Plenum consensus
  - Defines identity specific transactions.
  - https://github.com/Hyperledger/indy-node
- **Indy-SDK**
  - Provides APIs to applications for accessing Indy network
  - Indy- https://github.com/Hyperledger/indy-sdk

https://wiki.hyperledger.org/display/indy

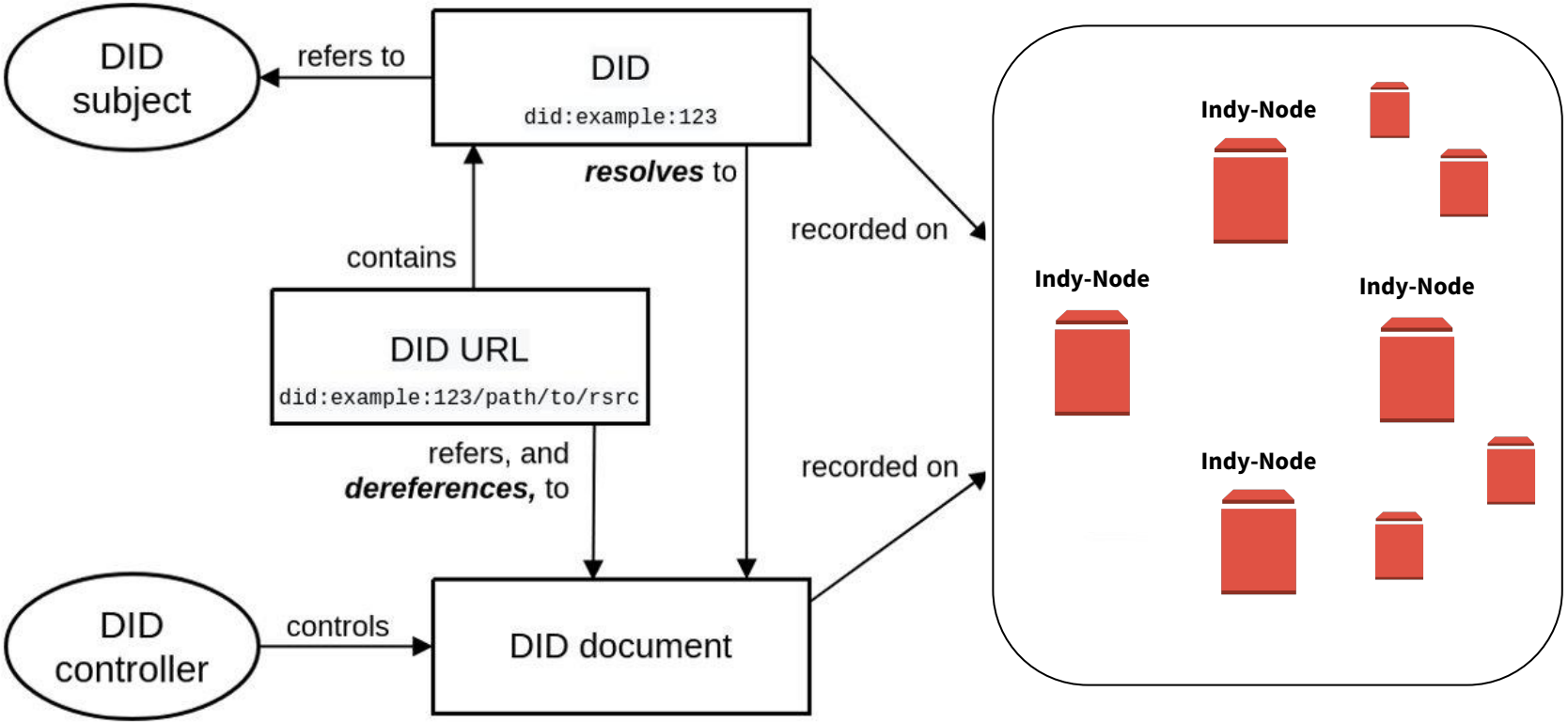
Indy and DIDs



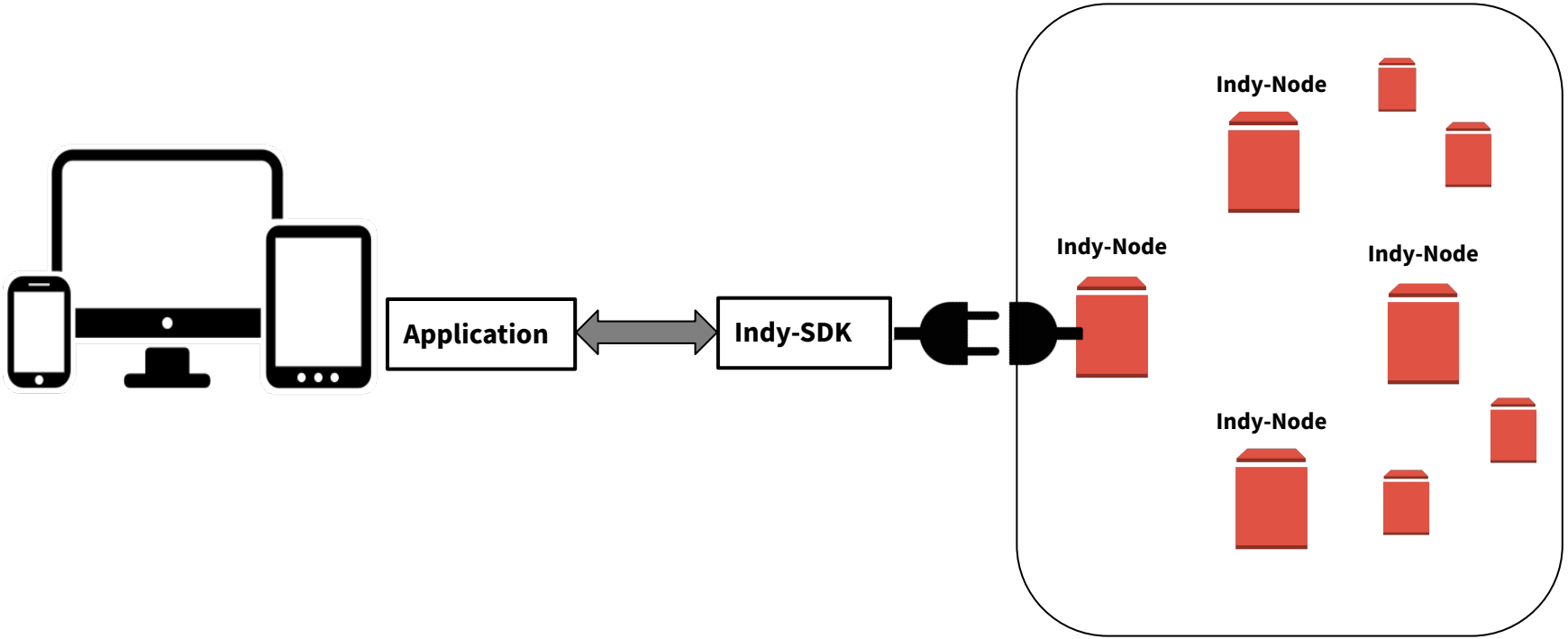
Indy and DIDs



Indy and DIDs



Indy and DIDs



Indy Projects

- **Indy-Plenum:**
  - Implements Byzantine Fault Tolerant Protocol
  - Used for consensus in Indy
  - Based on RBFT
  - https://github.com/Hyperledger/indy-plenum

- **Indy-Node:**
  - Implements the blockchain with Indy-Plenum consensus
  - Defines identity specific transactions.
  - https://github.com/Hyperledger/indy-node

- **Indy-SDK**
  - Provides APIs to applications for accessing Indy network
  - Indy- https://github.com/Hyperledger/indy-sdk

Start Indy Pool

Clone indy-sdk

```
git clone https://github.com/hyperledger/indy-sdk.git
```

```
cd indy-sdk
```

Build and run indy pool docker image

```
docker build -f ci/indy-pool.dockerfile -t indy_pool .
```

```
docker run -itd -p 9701-9708:9701-9708 indy_pool
```

Easier way to start indy pool

```
docker run -itd -p 9701-9708:9701-9708 ghoshbishakh/indy_pool
```

Alternatively start from indy-node

Clone indy-node

```
git clone https://github.com/hyperledger/indy-node.git
```

Move to the directory `indy-node/environment/docker/pool`

```
./pool_start.sh [number of nodes in pool] [IP addresses of nodes] [number of clients] [port for the first node]
```

Eg.

```
./pool_start.sh 4 10.0.0.2,10.0.0.3,10.0.0.4,10.0.0.5 10 9701
```

Install Indy SDK and wrappers

Ubuntu based distributions (Ubuntu 16.04 and 18.04)

It is recommended to install the SDK packages with APT:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys CE7709D068DB5E88

sudo add-apt-repository "deb https://repo.sovrin.org/sdk/deb (xenial|bionic) {release channel}"

sudo apt-get update

sudo apt-get install -y {library}
```

- {library} must be replaced with libindy, libnullpay, libvcx or indy-cli.
- (xenial|bionic) xenial for 16.04 Ubuntu and bionic for 18.04 Ubuntu.
- {release channel} must be replaced with master, rc or stable to define corresponded release channel. Please See the section "Release channels" above for more details.

Install Python3 Wrapper

```
pip install python3-indy
```

Connect to Indy Pool

Find path to **genesis txn**

Default pool genesis txn:

{"reqSignature":{},"txn":{"data":{"data":{"alias":"Node1","blskey":"4N8aUNHSgjQVgkpm8nhNEfDf6txHznoYREg9kirmJrkivgL4oSEimFF6nsQ6M41QvhM2Z33nves5vfSn9n1UwNFJBYtWVnHYMATn76vLuL3zU88KyeAYcHfsih3He6UHcXDxcaecHVz6jhCYz1P2UZn2bDVruL5wXpehgBfBaLKm3Ba","blskey_pop":"RahHYiCvoNCtPTrVtP7nMC5eTYrsUA8WjXbdhNc8debh1agE9bGiJxWBXYNFbnJXoXhWFMvyqhqhRoq737YQemH5ik9oL7R4NTTCz2LEZhkgLJzB3QRQqJyBNyv7acbdHrAT8nQ9UkLbaVL9NBpnWXBTw4LEMePaSHEw66RzPNdAX1","**client_ip":"127.0.0.1","client_port":9702,"node_ip":"127.0.0.1","node_port":9701**,"services":["VALIDATOR"]},"dest":"Gw6pDLhcBcoQesN72qfotTgFa7cbuqZpkX3Xo6pLhPhv"},"metadata":{"from":"Th7MpTaRZVRYnPiabds81Y"},"type":"0"},"txnMetadata":{"seqNo":1,"txnId":"fea82e10e894419fe2bea7d96296a6d46f50f93f9eeda954ec461b2ed2950b62"},"ver":"1"}

{"reqSignature":{},"txn":{"data":{"data":{"alias":"Node2","blskey":"37rAPpXVoxzKhz7d9gkUe52XuXryuLXoM6P6LbWDB7LSbG62Lsb33sfG7zqS8TK1MXwuCHj1FKNzVpsnafmqLG1vXN88rt38mNFs9TENzm4QHdBzsvCuoBnPH7rpYYDo9DZNJePaDvRvqJKByCabubJz3XXKbEeshzpz4Ma5QYpJqjk","blskey_pop":"Qr658mWZ2YC8JXGXwMDQTzuZCWF7NK9EwxphGmcBvCh6ybUuLxbG65nsX4JvD4SPNtkJ2w9ug1yLTj6fgmuDg41TgECXjLCij3RMsV8CwewBVgVN67wsA45DFWvqvLtu4rjNnE9JbdFTc1Z4WCPA3Xan44K1HoHAq9EVeaRYs8zoF5","client_ip":"127.0.0.1","client_port":9704,"node_ip":"127.0.0.1","node_port":9703,"services":["VALIDATOR"]},"dest":"8ECVSk179mjsjKRLWiQtssMLgp6EPhWXtaYyStWPSGAb"},"metadata":{"from":"EbP4aYNeTHL6q385GuVpRV"},"type":"0"},"txnMetadata":{"seqNo":2,"txnId":"1ac8aece2a18ced660fef8694b61aac3af08ba875ce3026a160acbc3a3af35fc"},"ver":"1"}

{"reqSignature":{},"txn":{"data":{"data":{"alias":"Node3","blskey":"3WFpdbg7C5cnLYZwFZevJqhubkFALBfCBBok15GdrKMUhUjGsk3jV6QKj6MZgEubF7oqCafxNdkm7eswgA4sdKTRc82tLGzZBd6vNqU8dupzup6uYUf32KTHTPQbuUM8Yk4QFXjEf2Usu2TJcNkdgpyeUSX42u5LqdDDpNSWUK5deC5","blskey_pop":"QwDeb2CkNSx6r8QC8vGQK3GRv7Yndn84TGNijX8YXHPiagXajyfTjoR87rXUu4G4QLk2cF8NNyqWiYMus1623dELWwx57rLCFqGh7N4ZRbGDRP4fnVcaKg1BcUxQ866Ven4gw8y4N56S5HzxXNBZtLYmhGHvDtk6PFkFwCvxYrNYjh","client_ip":"127.0.0.1","client_port":9706,"node_ip":"127.0.0.1","node_port":9705,"services":["VALIDATOR"]},"dest":"DKVxG2fXXTU8yT5N7hGEbXB3dfdAnYv1JczDUHpmDxya"},"metadata":{"from":"4cU41vWW82ArfxJxHkzXPG"},"type":"0"},"txnMetadata":{"seqNo":3,"txnId":"7e9f355dffa78ed24668f0e0e369fd8c224076571c51e2ea8be5f26479edebe4"},"ver":"1"}

{"reqSignature":{},"txn":{"data":{"data":{"alias":"Node4","blskey":"2zN3bHM1m4rLz54MJHYSwvqzPchYp8jkHswveCLAEJVcX6Mm1wHQD1SkPYMzUDTZvWvhuE6VNAkK3KxVeEmsanSmvjVkReDeBEMxeDaayjcZjFGPydyey1qxBHmTvAnBKoPydvuTAqx5f7YNNRAdeLmUi99gERUU7TD8KfAa6MpQ9bw","blskey_pop":"RPLagxaR5xdimFzwmzYnz4ZhWtYQEj8iR5ZU53T2gitPCyCHQneUn2Huc4oeLd2B2HzkGnjAff4hWTJT6C7qHYB1Mv2wU5iHHGFWkhnTX9WsEAbunJCV2qcaXScKj4tTfvdDKfLiVuU2av6hbsMztirRze7LvYBkRHV3tGwyCptsrP","client_ip":"127.0.0.1","client_port":9708,"node_ip":"127.0.0.1","node_port":9707,"services":["VALIDATOR"]},"dest":"4PS3EDQ3dW1tci1Bp6543CfuuebjFrg36kLAUcskGfaA"},"metadata":{"from":"TWwCRQRZ2ZHMJFn9TzLp7W"},"type":"0"},"txnMetadata":{"seqNo":4,"txnId":"aa5e817d7cc626170eca175822029339a444eb0ee8f0bd20d3b0b76e566fb008"},"ver":"1"}

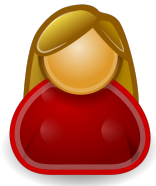
Connect to Pool

```
import asyncio
import json
from indy import anoncreds, did, ledger, pool, wallet, blob_storage
from indy.error import ErrorCode, IndyError


async def run():

    pool_ = {
        'name': 'pool1'
    }
    print("Open Pool Ledger: {}".format(pool_['name']))
    pool_['genesis_txn_path'] = "pool1.txn"
    pool_['config'] = json.dumps({"genesis_txn": str(pool_['genesis_txn_path'])})

    # Set protocol version 2 to work with Indy Node 1.4
    await pool.set_protocol_version(2)

    try:
        await pool.create_pool_ledger_config(pool_['name'], pool_['config'])
    except IndyError as ex:
        if ex.error_code == ErrorCode.PoolLedgerConfigAlreadyExistsError:
            pass
    pool_['handle'] = await pool.open_pool_ledger(pool_['name'], None)

    print(pool_['handle'])

loop = asyncio.get_event_loop()
loop.run_until_complete(run())
```

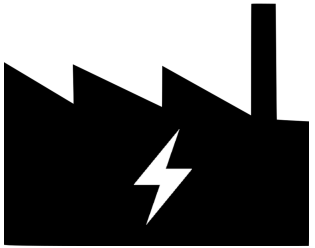
Path to genesis txn

Demo Scenario

University

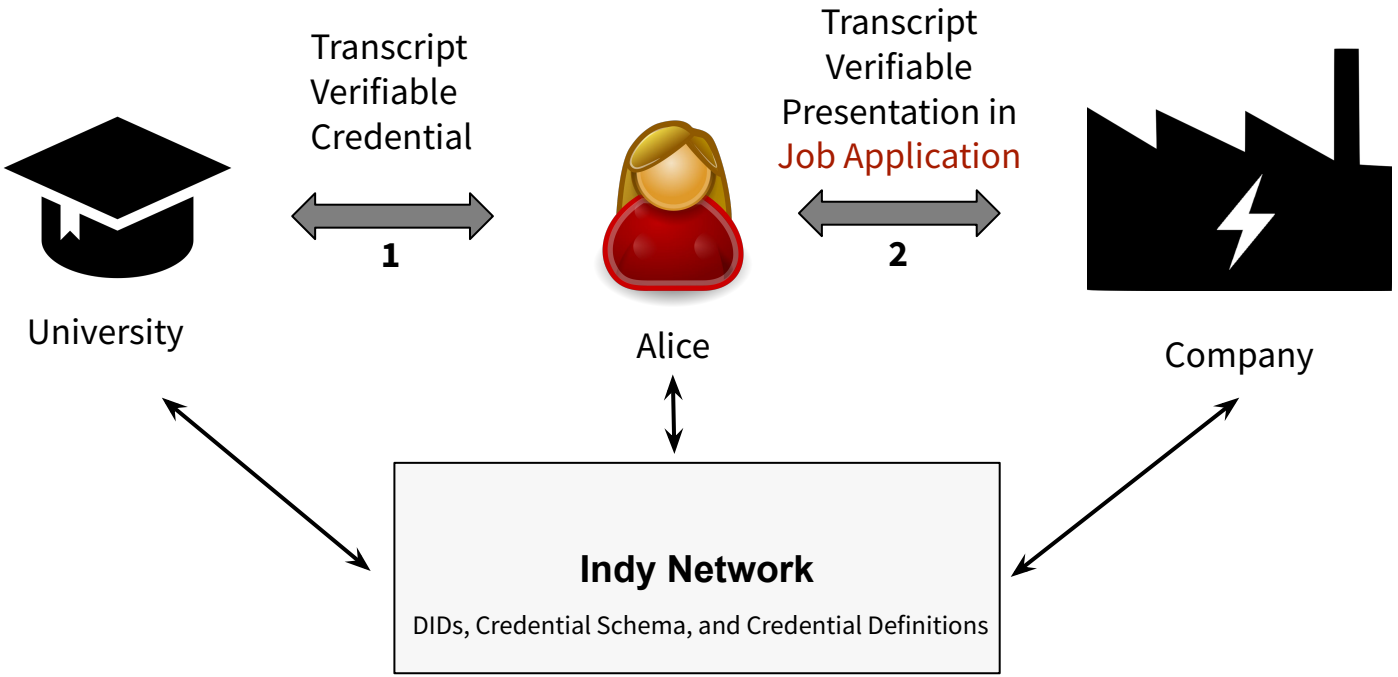


Alice



Company

Demo Scenario



Indy Roles

STEWARDS

Indy is designed to be operated such that everyone can see the contents of the blockchain (public), but only pre-approved participants, known as **stewards**, are permitted to participate in the validation process (permissioned).

Trust Anchor(TA)

TA's are the link between User and Stewards. TA can be banks, universities, hospitals, service providers, insurance companies. TA's are onboarded by approvals of Stewards. So TA accepts the request from user and forwards this request to Stewards in case of writing into the ledger.

Demo Setup

- Access the Stewards, as well as the parties - Alice, University, and Company from the same code and the same wallet.

- In a real scenario, these will be different applications running independently with each party.

- The indy pool nodes are also running in a single host. In a real scenario the indy pool will be formed of independent stakeholders of the identity network. Eg. Sovrin network.

Step 1: Getting the ownership for Steward's DID

```
async def create_wallet(identity):
    print("\"{}\" -> Create wallet".format(identity['name']))
    try:
        await wallet.create_wallet(identity['wallet_config'],
                                   identity['wallet_credentials'])
    except IndyError as ex:
        if ex.error_code == ErrorCode.PoolLedgerConfigAlreadyExistsError:
            pass
    identity['wallet'] = await wallet.open_wallet(identity['wallet_config'],
                                                  identity['wallet_credentials'])

# ================ Step 1 - getting steward verinym ownership ------------------------------
steward = {
        'name': "Sovrin Steward",
        'wallet_config': json.dumps({'id': 'sovrin_steward_wallet'}),
        'wallet_credentials': json.dumps({'key': 'steward_wallet_key'}),
        'pool': pool_['handle'],
        'seed': '000000000000000000000000Steward1'
    }

await create_wallet(steward)
```

Step 2: Register DID for Government, University and Company

```
government = {
        'name': 'Government',
        'wallet_config': json.dumps({'id': 'government_wallet'}),
        'wallet_credentials': json.dumps({'key': 'government_wallet_key'}),
        'pool': pool_['handle'],
        'role': 'TRUST_ANCHOR'
    }

await getting_verinym(steward, government)
```

Verinyms are DIDs registered through Stewards, where stewards act as trust anchors attesting their identity.

NYM Request

```
async def getting_verinym(from_, to):
    await create_wallet(to)

    (to['did'], to['key']) = await did.create_and_store_my_did(to['wallet'], "{}")

    from_['info'] = {
        'did': to['did'],
        'verkey': to['key'],
        'role': to['role'] or None
    }

    await send_nym(from_['pool'], from_['wallet'], from_['did'], from_['info']['did'],
                   from_['info']['verkey'], from_['info']['role'])


async def send_nym(pool_handle, wallet_handle, _did, new_did, new_key, role):
    nym_request = await ledger.build_nym_request(_did, new_did, new_key, None, role)
    print(nym_request)
    await ledger.sign_and_submit_request(pool_handle, wallet_handle, _did, nym_request)
```

Step 3: Government creates credential schema for university transcript

- **Credential Schema** is the base semantic structure that describes the list of attributes which one particular Credential can contain.
- Note: It's not possible to update an existing Schema. So, if the Schema needs to be evolved, a new Schema with a new version or name needs to be created.
- A Credential Schema can be created and saved in the Ledger by any **Trust Anchor**.

Step 3: Government creates credential schema for university transcript

```
transcript = {
    'name': 'Transcript',
    'version': '1.2',
    'attributes': ['first_name', 'last_name', 'degree', 'status', 'year', 'average', 'ssn']
}
(government['transcript_schema_id'], government['transcript_schema']) = \
    await anoncreds.issuer_create_schema(government[ 'did'], transcript['name'], transcript['version'],
                                         json.dumps(transcript[ 'attributes']))
transcript_schema_id = government[ 'transcript_schema_id']


schema_request = await ledger.build_schema_request(government['did'], government['transcript_schema'])
await ledger.sign_and_submit_request(government['pool'], government['wallet'], government['did'], schema_request)
```

Step 4: University registers credential definition

- **Credential Definition** specifies the **keys that the Issuer uses for the signing of Credentials** , and it also states a specific Credential Schema that the credential will follow.
- Note It's not possible to update data in an existing Credential Definition. So, if a `CredDef` needs to be evolved (for example, a key needs to be rotated), then a new Credential Definition needs to be created by a new Issuer DID.
- A Credential Definition can be created and saved in the Ledger by any Trust Anchor. Here University creates and publishes a Credential Definition for the known Transcript Credential Schema to the Ledger.

Step 4: University registers credential definition

```
# GET SCHEMA FROM LEDGER
get_schema_request = await ledger.build_get_schema_request(theUniversity[ 'pool'], transcript_schema_id)
get_schema_response = await ensure_previous_request_applied(
    theUniversity['did'], get_schema_request, lambda response: response['result']['data'] is not None)
(theUniversity[ 'transcript_schema_id' ], theUniversity[ 'transcript_schema' ]) = await ledger.parse_get_schema_response(get_schema_response)

# CREATE TRANSCRIPT CREDENTIAL DEFINITION
transcript_cred_def = {
    'tag': 'TAG1',
    'type': 'CL',
    'config': {"support_revocation": False}
}
(theUniversity['transcript_cred_def_id'], theUniversity['transcript_cred_def']) = \
    await anoncreds.issuer_create_and_store_credential_def(theUniversity[ 'wallet'], theUniversity['did'],
                                                           theUniversity[ 'transcript_schema'], transcript_cred_def['tag'],
                                                           transcript_cred_def[ 'type'],
                                                           json.dumps(transcript_cred_def[ 'config']))

# COMMIT CREDENTIAL DEFINITION TO LEDGER
cred_def_request = await ledger.build_cred_def_request(theUniversity[ 'did'], theUniversity['transcript_cred_def'])
await ledger.sign_and_submit_request(theUniversity[ 'pool'], theUniversity['wallet'], theUniversity['did'], cred_def_request)
```

Step 5: Alice receives Transcript VC from University

- Setup Alice's Wallet
- University creates a **Transcript Credential Offer**
  - Contains Schema, Credential Definition
- Alice creates **Transcript Credential Request**
  - Contains a **Master Secret**
- University issues the Credential
- Credential is transferred from University to Alice.

Note: A Master Secret is an item of Private Data used by a Holder to guarantee that a credential uniquely applies to them. The Master Secret is an input that combines data from multiple Credentials to prove that the Credentials have a common subject (the Holder). A Master Secret should be known only to the Holder.

Setup Alice's Wallet

```
alice = {
       'name': 'Alice',
       'wallet_config': json.dumps({'id': 'alice_wallet'}),
       'wallet_credentials': json.dumps({'key': 'alice_wallet_key'}),
       'pool': pool_['handle'],
   }
await create_wallet(alice)
(alice['did'], alice['key']) = await did.create_and_store_my_did(alice['wallet'], "{}")
```

University sends credential offer to Alice

```
theUniversity['transcript_cred_offer'] = \
       await anoncreds.issuer_create_credential_offer(theUniversity['wallet'],
theUniversity['transcript_cred_def_id'])


print("\"theUniversity\" -> Send \"Transcript\" Credential Offer to Alice")
alice['transcript_cred_offer'] = theUniversity['transcript_cred_offer']
transcript_cred_offer_object = json.loads(alice['transcript_cred_offer'])
```

Alice sends credential request to University

```
print("\"Alice\" -> Create and store \"Alice\" Master Secret in Wallet")
alice['master_secret_id'] = await anoncreds.prover_create_master_secret(alice['wallet'], None)

print("\"Alice\" -> Get \"theUniversity Transcript\" Credential Definition from Ledger")
(alice['theUniversity_transcript_cred_def_id'], alice['theUniversity_transcript_cred_def']) = \
   await get_cred_def(alice['pool'], alice['did'], alice['transcript_cred_def_id'])

print("\"Alice\" -> Create \"Transcript\" Credential Request for theUniversity")
(alice['transcript_cred_request'], alice['transcript_cred_request_metadata']) = \
   await anoncreds.prover_create_credential_req(alice['wallet'], alice['did'],
                                                alice['transcript_cred_offer'],
                                                alice['theUniversity_transcript_cred_def'],
                                                alice['master_secret_id'])

print("\"Alice\" -> Send \"Transcript\" Credential Request to theUniversity")
theUniversity['transcript_cred_request'] = alice['transcript_cred_request']
```

University issues Transcript Credential for Alice

```
print("\"theUniversity\" -> Create \"Transcript\" Credential for Alice" )
theUniversity['alice_transcript_cred_values'] = json.dumps({
   "first_name": {"raw": "Alice", "encoded": "1139481716457488690172217916278103335"},
   "last_name": {"raw": "Garcia", "encoded": "53216427802417901235879024567891234 52"},
   "degree": {"raw": "Bachelor of Science, Marketing", "encoded": "12434523576212321"},
   "status": {"raw": "graduated", "encoded": "2213454313412354"},
   "ssn": {"raw": "123-45-6789", "encoded": "3124141231422543541"},
   "year": {"raw": "2015", "encoded": "2015"},
   "average": {"raw": "5", "encoded": "5"}
})
theUniversity['transcript_cred'], _, _ = \
   await anoncreds.issuer_create_credential(theUniversity[ 'wallet'], theUniversity['transcript_cred_offer'],
                                            theUniversity[ 'transcript_cred_request'],
                                            theUniversity[ 'alice_transcript_cred_values'], None, None)

print("\"theUniversity\" -> Send \"Transcript\" Credential to Alice" )
alice['transcript_cred'] = theUniversity['transcript_cred']

print("\"Alice\" -> Store \"Transcript\" Credential from theUniversity" )
_, alice['transcript_cred_def'] = await get_cred_def(alice['pool'], alice['did'],
                                                     alice[ 'transcript_cred_def_id'])

await anoncreds.prover_store_credential(alice[ 'wallet'], None, alice['transcript_cred_request_metadata'],
                                        alice[ 'transcript_cred'], alice['transcript_cred_def'], None)
```

Step 6 - Verifiable Presentation:

- Company creates job application proof format
- Company sends job application proof request to Alice
- Alice fetches transcript credentials from wallet
- Alice prepares the verifiable presentation
  - Claims are split into two types:
    - Revealed attributes
    - Zero Knowledge proofs for predicates
- Alice sends job application presentation to Company
- Company verifies values and predicates in presentation

Job application proof request

```
theCompany['job application_proof_request'] = json.dumps({
   'nonce': nonce,
   'name': 'Job-Application',
   'version': '0.1',
   'requested_attributes': {
       'attr1 referent': {
           'name': 'first_name'
       },
       'attr2 referent': {
           'name': 'last_name'
       },
       'attr3 referent': {
           'name': 'degree',
           'restrictions': [{'cred_def_id': theUniversity['transcript_cred_def_id']}]
       },
       'attr4 referent': {
           'name': 'status',
           'restrictions': [{'cred_def_id': theUniversity['transcript_cred_def_id']}]
       },
       'attr5_referent': {
           'name': 'ssn',
           'restrictions': [{'cred_def_id': theUniversity['transcript_cred_def_id']}]
       },
       'attr6 referent': {
           'name': 'phone_number'
       }
   },
   'requested predicates': {
       'predicate1 referent': {
           'name': 'average',
           'p type': '>=',
           'p value': 4,
           'restrictions': [{'cred_def_id': theUniversity['transcript_cred_def_id']}]
       }
   }
})
```

Preparing Verifiable Presentation

```
alice['job_application_requested_creds'] = json.dumps({
    'self_attested_attributes': {
        'attr1_referent': 'Alice',
        'attr2_referent': 'Garcia',
        'attr6_referent': '123-45-6789'
    },
    'requested_attributes': {
        'attr3_referent': {'cred_id': cred_for_attr3['referent'], 'revealed': True},
        'attr4_referent': {'cred_id': cred_for_attr4['referent'], 'revealed': True},
        'attr5_referent': {'cred_id': cred_for_attr5['referent'], 'revealed': True},
    },
    'requested_predicates': {'predicate1_referent': {'cred_id': cred_for_predicate1['referent']}}
})
```

References

Indy Walkthrough:
https://github.com/hyperledger/indy-sdk/blob/master/docs/getting-started/indy-walkthrough.md

Indy Walkthrough Python Code:
https://github.com/hyperledger/indy-sdk/blob/master/samples/python/src/getting_started.py

Sample code in other languages:
https://github.com/hyperledger/indy-sdk/tree/master/samples

Indy-node: https://github.com/hyperledger/indy-node

Indy-sdk: https://github.com/hyperledger/indy-sdk

Indy-plenum: https://github.com/hyperledger/indy-plenum