Getting Started with Libindy

A Developer Guide for Building Indy Clients Using Libindy

What Indy and Libindy are and Why They Matter

Indy provides a software ecosystem for private, secure, and powerful identity, and libindy enables clients for it. Indy puts people — not the organizations that traditionally centralize identity — in charge of decisions about their own privacy and disclosure. This enables all kinds of rich innovation: connection contracts, revocation, novel payment workflows, asset and document management features, creative forms of escrow, curated reputation, integrations with other cool technologies, and so on.

Indy uses open-source, distributed ledger technology. These ledgers are a form of database that is provided cooperatively by a pool of participants, instead of by a giant database with a central admin. Data lives redundantly in many places, and it accrues in transactions orchestrated by many machines. Strong, industry-standard cryptography protects it. Best practices in key management and cybersecurity pervade its design. The result is a reliable, public source of truth under no single entity's control, robust to system failure, resilient to hacking, and highly immune to subversion by hostile entities.

If the concepts of cryptography and blockchain details feel mysterious, fear not: this guide will help introduce you to key concepts within Indy. You're starting in the right place.

What We'll Cover

Our goal is to introduce you to many of the concepts of Indy and give you some idea of what happens behind the scenes to make it all work.

We're going to frame the exploration with a story. Alice, a graduate of the fictional Faber College, wants to apply for a job at the fictional company Acme Corp. As soon as she has the job, she wants to apply for a loan in Thrift Bank so she can buy a car. She would like to use her college transcript as proof of her education on the job application and once hired, Alice would like to use the fact of employment as evidence of her creditworthiness for the loan.

The sorts of identity and trust interactions required to pull this off are messy in the world today; they are slow, they violate privacy, and they are susceptible to fraud. We'll show you how Indy is a quantum leap forward.

Ready?

About Alice

As a graduate of Faber College, Alice receives an alumni newsletter where she learns that her alma mater is offering digital transcripts. She logs in to the college alumni website and requests her transcript by clicking **Get Transcript**. (Other ways to initiate this request might include scanning a QR code, downloading a transcript package from a published URL, etc.)

Alice doesn't realize it yet, but to use this digital transcript she will need a new type of identity — not the traditional identity that Faber College has built for her in its on-campus database, but a new and portable one that belongs to her, independent of all past and future relationships, that nobody can revoke or co-opt or correlate without her permission. This is a ***self-sovereign identity*** and it is the core feature of Indy.

In normal contexts, managing a self-sovereign identity will require a tool such as a desktop or mobile application. It might be a standalone app or it might leverage a third party service provider that the ledger calls an **agency**. The Sovrin Foundation publishes reference versions of such tools. Faber College will have studied these requirements and will recommend an ***Indy app*** to Alice if she doesn't already have one. This app will install as part of the workflow from the **Get Transcript** button.

When Alice clicks **Get Transcript**, she will download a file that holds an Indy **connection request**. This connection request file, having an .indy extension and associated with her Indy app, will allow her to establish a secure channel of communication with another party in the ledger ecosystem — Faber College.

So when Alice clicks **Get Transcript**, she will normally end up installing an app (if needed), launching it, and then being asked by the app whether she wants to accept a request to connect with Faber.

For this guide, however, we'll be using an **Indy SDK API** (as provided by libindy) instead of an app, so we can see what happens behind the scenes. We will pretend to be a particularly curious and technically adventurous Alice…

Infrastructure Preparation

Step 1: Getting Trust Anchor Credentials for Faber, Acme, Thrift and Government

Faber College and other actors have done some preparation to offer this service to Alice. To understand these steps let's start with some definitions.

The ledger is intended to store **Identity Records** that describe a **Ledger Entity**. Identity Records are public data and may include Public Keys, Service Endpoints, Credential Schemas, and Credential Definitions. Every **Identity Record** is associated with exactly one **DID** (Decentralized Identifier) that is globally unique and resolvable (via a ledger) without requiring any centralized resolution authority. To maintain privacy each **Identity Owner** can own multiple DIDs.

In this tutorial we will use two types of DIDs. The first one is a **Verinym**. A **Verinym** is associated with the **Legal Identity** of the **Identity Owner**. For example, all parties should be able to verify that some DID is used by a Government to publish schemas for some document type. The second type is a **Pseudonym** - a **Blinded Identifier** used to maintain privacy in the context of an ongoing digital relationship (**Connection**). If the Pseudonym is used to maintain only one digital relationship we will call it a Pairwise-Unique Identifier. We will use Pairwise-Unique Identifiers to maintain secure connections between actors in this tutorial.

The creation of a DID known to the Ledger is an **Identity Record** itself (NYM transaction). The NYM transaction can be used for creation of new DIDs that is known to that ledger, the setting and rotation of a verification key, and the setting and changing of roles. The most important fields of this transaction are `dest` (target DID), `role` (role of a user NYM record being created for) and the `verkey` (target verification key). See Requests to get more information about supported ledger transactions.

Publishing with a DID verification key allows a person, organization or thing, to verify that someone owns this DID as that person, organization or thing is the only one who knows the corresponding signing key and any DID-related operations requiring signing with this key.

Our ledger is public permissioned and anyone who wants to publish DIDs needs to get the role of **Trust Anchor** on the ledger. A **Trust Anchor** is a person or organization that the ledger already knows about, that is able to help bootstrap others. (It is *not* the same as what cybersecurity experts call a "trusted third party"; think of it more like a facilitator). See Roles to get more information about roles.

The first step towards being able to place transactions on the ledger involves getting the role of Trust Anchor on the ledger. Faber College, Acme Corp and Thrift Bank will need to get the role of Trust Anchor on the ledger so they can create Verinyms and Pairwise-Unique Identifiers to provide the service to Alice.

Becoming a **Trust Anchor** requires contacting a person or organization who already has the **Trust Anchor** role on the ledger. For the sake of the demo, in our empty test ledger we have only NYMs with the **Steward** role, but all **Stewards** are automatically **Trust Anchors**.

Step 2: Connecting to the Indy Nodes Pool

We are ready to start writing the code that will cover Alice's use case from start to finish. It is important to note that for demo purposes it will be a single test that will contain the code intended to be executed on different agents. We will always point to what Agent is intended to execute each code part. Also we will use different wallets to store the DID and keys of different Agents. Let's begin.

The first code block will contain the code of the **Steward's** agent.

To write and read the ledger's transactions after gaining the proper role, you'll need to make a connection to the Indy nodes pool. To make a connection to the different pools that exist, like the Sovrin pool or the local pool we started by ourselves as part of this tutorial, you'll need to set up a pool configuration.

The list of nodes in the pool is stored in the ledger as NODE transactions. Libindy allows you to restore the actual list of NODE transactions by a few known transactions that we call genesis transactions. Each **Pool Configuration** is defined as a pair of pool configuration name and pool configuration JSON. The most important field in pool configuration json is the path to the file with the list of genesis transactions. Make sure this path is correct.

The `pool.create_pool_ledger_config` call allows you to create a named pool configuration. After the pool configuration is created we can connect to the nodes pool that this configuration describes by calling `pool.open_pool_ledger`. This call returns the pool handle that can be used to reference this opened connection in future libindy calls.

The code block below contains each of these items. Note how the comments denote that this is the code for the "Steward Agent."

```
# Steward Agent
pool_name = 'pool1'
pool_genesis_txn_path = get_pool_genesis_txn_path(pool_name)
pool_config = json.dumps({"genesis_txn": str(pool_genesis_txn_path)})
await pool.create_pool_ledger_config(pool_name, pool_config)
pool_handle = await pool.open_pool_ledger(pool_name, None)
```

Step 3: Getting the ownership for Steward's Verinym

Next, the Steward's agent should get the ownership for the DID that has corresponding NYM transactions with the Steward role on the ledger.

The test ledger we use was pre-configured to store some known **Steward** NYMs. Also we know **seed** values for the random number generator that were used to generate keys for these NYMs. These **seed** values allow us to restore signing keys for these DIDs on the **Steward's** agent side and as a result get the DID ownership.

Libindy has a concept of the **Wallet**. The wallet is secure storage for crypto materials like DIDs, keys, etc... To store the **Steward's** DID and corresponding signkey, the agent should create a named wallet first by calling `wallet.create_wallet`. After this the named wallet can be opened by calling `wallet.open_wallet`. This call returns the wallet handle that can be used to reference this opened wallet in future libindy calls.

After the wallet is opened we can create a DID record in this wallet by calling `did.create_and_store_my_did` that returns the generated DID and verkey part of the generated key. The signkey part for this DID will be stored in the wallet too, but it is impossible to read it directly.

```
# Steward Agent
steward_wallet_name = 'sovrin_steward_wallet'
await wallet.create_wallet(pool_name, steward_wallet_name, None, None, None)
steward_wallet = await wallet.open_wallet(steward_wallet_name, None, None)

steward_did_info = {'seed': '000000000000000000000000Steward1'}
(steward_did, steward_key) = await did.create_and_store_my_did(steward_wallet, json.dumps(steward_did_info))
```

Please note: We provided only information about the seed to `did.create_and_store_my_did`, but not any information about the Steward's DID. By default DID's are generated as the first 16 bytes of the verkey. For such DID's, when dealing with operations that require both a DID and the verkey we can use the verkey in an abbreviated form. In this form the verkey starts with a tilde '~' followed by 22 or 23 characters. The tilde indicates that the DID itself represents the first 16 bytes of the verkey and the string following the tilde represents the second 16 bytes of the verkey, both using base58Check encoding.

Step 4: Onboarding Faber, Acme, Thrift and Government by Steward

Faber, Acme, Thrift and Government should now establish a Connection with the Steward.

Each connection is actually a pair of Pairwise-Unique Identifiers (DIDs). The one DID is owned by one party to the connection and the second by another.

Both parties know both DIDs and understand what connection this pair describes.

The relationship between them is not shareable with others; it is unique to those two parties in that each pairwise relationship uses different DIDs.

We call the process of establish a connection **Onboarding**.

In this tutorial we will describe the simple version of onboarding process. In our case, one party will always be the Trust Anchor. Real enterprise scenarios can use a more complex version.

Connecting the Establishment

Let's look the process of connection establishment between **Steward** and **Faber College**.

1. **Faber** and **Steward** contact in some way to initiate onboarding process. It can be filling the form on web site or a phone call.

2. **Steward** creates a new DID record in the wallet by calling `did.create_and_store_my_did` that he will use for secure interactions only with **Faber**.

```
# Steward Agent
(steward_faber_did, steward_faber_key) = await did.create_and_store_my_did(steward_wallet, "{}")
```

3. **Steward** sends the corresponding `NYM` transaction to the Ledger by consistently calling `ledger.build_nym_request` to build the NYM request and `ledger.sign_and_submit_request` to send the created request.

```
# Steward Agent
nym_request = await ledger.build_nym_request(steward_did, steward_faber_did, steward_faber_key, None, role)
await ledger.sign_and_submit_request(pool_handle, steward_wallet, steward_did, nym_request)
```

4. **Steward** creates the connection request which contains the created `DID` and `Nonce`. This nonce is just a big random number generated to track the unique connection request. A nonce is a random arbitrary number that can only be used one time. When a connection request is accepted, the invitee digitally signs the nonce so that the inviter can match the response with a prior request.

```
# Steward Agent
connection_request = {
    'did': steward_faber_did,
    'nonce': 123456789
}
```

5. **Steward** sends the connection request to **Faber**.

6. **Faber** accepts the connection request from **Steward**.

7. **Faber** creates a wallet if it does not exist yet.

```
# Faber Agent
await wallet.create_wallet(pool_name, 'faber_wallet', None, None, None)
faber_wallet = await wallet.open_wallet('faber_wallet', None, None)
```

8. **Faber** creates a new DID record in its wallet by calling `did.create_and_store_my_did` that it will use only for secure interactions with the **Steward**.

```
# Faber Agent
(faber_steward_did, faber_steward_key) = await did.create_and_store_my_did(faber_wallet, "{}")
```

9. **Faber** creates the connection response which contains the created `DID`, `Verkey` and `Nonce` from the received connection request.

```
# Faber Agent
connection_response = json.dumps({
    'did': faber_steward_did,
    'verkey': faber_steward_key,
```

```
        'nonce': connection_request['nonce']
    })
```

10. **Faber** asks the ledger for the Verification key of the **Steward's** DID by calling `did.key_for_did`.

```
# Faber Agent
steward_faber_verkey = await did.key_for_did(pool_handle, faber_wallet, connection_request['did'])
```

11. **Faber** anonymously encrypts the connection response by calling `crypto.anon_crypt` with the **Steward** verkey. 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. While the Recipient can verify the integrity of the message, it cannot verify the identity of the Sender.

```
# Faber Agent
anoncrypted_connection_response = await crypto.anon_crypt(steward_faber_verkey, connection_response.encode('utf-8
```

12. **Faber** sends the anonymously encrypted connection response to the **Steward**.

13. **Steward** anonymously decrypts the connection response by calling `crypto.anon_decrypt`.

```
# Steward Agent
decrypted_connection_response = \
    (await crypto.anon_decrypt(steward_wallet, steward_faber_key, anoncrypted_connection_response)).decode("utf-8
```

14. **Steward** authenticates **Faber** by the comparison of Nonce.

```
# Steward Agent
assert connection_request['nonce'] == decrypted_connection_response['nonce']
```

15. **Steward** sends the `NYM` transaction for **Faber's** DID to the Ledger. Please note that despite the fact that the Steward is the sender of this transaction the owner of the DID will be Faber as it uses the verkey as provided by Faber.

```
# Steward Agent
nym_request = await ledger.build_nym_request(steward_did, decrypted_connection_response['did'], decrypted_connect
await ledger.sign_and_submit_request(pool_handle, steward_wallet, steward_did, nym_request)
```

At this point **Faber** is connected to the **Steward** and can interact in a secure peer-to-peer way. **Faber** can trust the response is from **Steward** because:

- it connects to the current endpoint
- no replay - attack is possible, due to her random challenge
- it knows the verification key used to verify **Steward** digital signature is the correct one because it just confirmed it on the ledger

Note: All parties must not use the same DID's to establish other relationships. By having independent pairwise relationships, you're reducing the ability for others to correlate your activities across multiple interactions.

Getting Verinym

It is important to understand that earlier created **Faber** DID is not, in and of itself, the same thing as self-sovereign identity. This DID must be used only for secure interaction with the **Steward**. After the connection is established **Faber** must create new DID record that he will use as Verinym in the Ledger.

1. **Faber** creates a new DID in its wallet by calling `did.create_and_store_my_did`.

```
# Faber Agent
(faber_did, faber_key) = await did.create_and_store_my_did(faber_wallet, "{}")
```

2. **Faber** prepares the message that will contain the created DID and verkey.

```
# Faber Agent
faber_did_info_json = json.dumps({
    'did': faber_did,
    'verkey': faber_key
})
```

3. **Faber** authenticates and encrypts the message by calling `crypto.auth_crypt` function, which is an implementation of the authenticated-encryption schema. Authenticated encryption 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.

```
# Faber Agent
authcrypted_faber_did_info_json = \
    await crypto.auth_crypt(faber_wallet, faber_steward_key, steward_faber_key, faber_did_info_json.encode('utf-8
```

4. **Faber** sends the encrypted message to the **Steward**.

5. **Steward** decrypts the received message by calling `crypto.auth_decrypt` .

```
# Steward Agent
sender_verkey, authdecrypted_faber_did_info_json = \
    await crypto.auth_decrypt(steward_handle, steward_faber_key, authcrypted_faber_did_info_json)
faber_did_info = json.loads(authdecrypted_faber_did_info_json)
```

6. **Steward** asks the ledger for the Verification key of **Faber's** DID by calling `did.key_for_did` .

```
# Steward Agent
faber_verkey = await did.key_for_did(pool_handle, from_wallet, faber_did_info['did'])
```

7. **Steward** authenticates **Faber** by comparison of the Message Sender Verkey and the **Faber** Verkey received from the Ledger.

```
# Steward Agent
assert sender_verkey == faber_verkey
```

8. **Steward** sends the corresponded NYM transaction to the Ledger with `TRUST ANCHOR` role. Please note that despite the fact that the Steward is the sender of this transaction the owner of DID will be Faber as it uses Verkey provided by Faber.

```
# Steward Agent
nym_request = await ledger.build_nym_request(steward_did, decrypted_faber_did_info_json['did'],
                                             decrypted_faber_did_info_json['verkey'], None, 'TRUST_ANCHOR')
await ledger.sign_and_submit_request(pool_handle, steward_wallet, steward_did, nym_request)
```

At this point **Faber** has a DID related to his identity in the Ledger.

Acme, **Thrift Bank**, and **Government** must pass the same Onboarding process connection establishment with **Steward**.

Step 5: Credential Schemas Setup

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**.

Here is where the **Government** creates and publishes the **Transcript** Credential Schema to the Ledger:

1. The **Trust Anchor** creates the **Credential Schema** by calling the `anoncreds.issuer_create_schema` that returns the generated **Credential Schema**.

```
# Government Agent
(transcript_schema_id, transcript_schema) = \
    await anoncreds.issuer_create_schema(government_did,
                                         'Transcript',
                                         '1.2',
                                         json.dumps(['first_name', 'last_name', 'degree', 'status', 'year', 'aver
```

2. The **Trust Anchor** sends the corresponding Schema transaction to the Ledger by consistently calling the `ledger.build_schema_request` to build the Schema request and `ledger.sign_and_submit_request` to send the created request.

```
# Government Agent
schema_request = await ledger.build_schema_request(government_did, transcript_schema)
await ledger.sign_and_submit_request(pool_handle, government_wallet, government_did, schema_request)
```

In the same way **Government** creates and publishes the **Job-Certificate** Credential Schema to the Ledger:

```
# Government Agent
  (job_certificate_schema_id, job_certificate_schema) = \
      await anoncreds.issuer_create_schema(government_did,
                                           'Job-Certificate',
                                           '0.2',
                                           json.dumps(['first_name', 'last_name', 'salary', 'employee_status', 'exp
schema_request = await ledger.build_schema_request(government_did, json.dumps(to the Ledger))
await ledger.sign_and_submit_request(pool_handle, government_wallet, government_did, schema_request)
```

At this point we have the **Transcript** and the **Job-Certificate** Credential Schemas published by **Government** to the Ledger.

Step 6: Credential Definition Setup

Credential Definition is similar in that the keys that the Issuer uses for the signing of Credentials also satisfies a specific Credential Schema.

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 **Faber** creates and publishes a Credential Definition for the known **Transcript** Credential Schema to the Ledger.

1. The **Trust Anchor** gets the specific **Credential Schema** from the Ledger by consistently calling the `ledger.build_get_schema_request` to build the `GetSchema` request, `ledger.sign_and_submit_request` to send the created request and the `ledger.parse_get_schema_response` to get the `Schema` in the format required by Anoncreds API from the `GetSchema` response.

```
# Faber Agent
get_schema_request = await ledger.build_get_schema_request(faber_did, transcript_schema_id)
get_schema_response = await ledger.submit_request(pool_handle, get_schema_request)
(transcript_schema_id, transcript_schema) = await ledger.parse_get_schema_response(get_schema_response)
```

2. The **Trust Anchor** creates the **Credential Definition** related to the received **Credential Schema** by calling `anoncreds.issuer_create_and_store_credential_def` that returns the generated public **Credential Definition**. The private Credential Definition part for this **Credential Schema** will be stored in the wallet too, but it is impossible to read it directly.

```
# Faber Agent
(faber_transcript_cred_def_id, faber_transcript_cred_def_json) = \
    await anoncreds.issuer_create_and_store_credential_def(faber_wallet, faber_did, transcript_schema, 'TAG1', '(
```

3. The **Trust Anchor** sends the corresponding `CredDef` transaction to the Ledger by consistently calling the `ledger.build_cred_def_request` to build the `CredDef` request and the `ledger.sign_and_submit_request` to send the created request.

```
# Faber Agent
cred_def_request = await ledger.build_cred_def_request(faber_did, faber_transcript_cred_def_json)
await ledger.sign_and_submit_request(pool_handle, faber_wallet, faber_did, cred_def_request)
```

The same way **Acme** creates and publishes a **Credential Definition** for the known **Job-Certificate** Credential Schema to the Ledger.

```
# Acme Agent
get_schema_request = await ledger.build_get_schema_request(acme_did, job_certificate_schema_id)
get_schema_response = await ledger.submit_request(pool_handle, get_schema_request)
(job_certificate_schema_id, job_certificate_schema) = await ledger.parse_get_schema_response(get_schema_response)

(acme_job_certificate_cred_def_id, acme_job_certificate_cred_def_json) = \
    await anoncreds.issuer_create_and_store_credential_def(acme_wallet, acme_did, job_certificate_schema, 'TAG1', '

  cred_def_request = await ledger.build_cred_def_request(acme_did, acme_job_certificate_cred_def_json)
  await ledger.sign_and_submit_request(pool_handle, acme_wallet, acme_did, cred_def_request)
```

At this point we have a **Credential Definition** for the **Job-Certificate** Credential Schema published by **Acme** and a **Credential Definition** for the **Transcript** Credential Schema published by **Faber**.

Alice Gets a Transcript

A credential is a piece of information about an identity — a name, an age, a credit score... It is information claimed to be true. In this case, the credential is named, "Transcript".

Credentials are offered by an issuer.

An issuer may be any identity owner known to the Ledger and any issuer may issue a credential about any identity owner it can identify.

The usefulness and reliability of a credential are tied to the reputation of the issuer with respect to the credential at hand. For Alice to self-issue a credential that she likes chocolate ice cream may be perfectly reasonable, but for her to self-issue a credential that she graduated from Faber College should not impress anyone.

As we mentioned in About Alice, **Alice** graduated from **Faber College**. After **Faber College** had established a connection with Alice, it created for her a Credential Offer about the issuance of the **Transcript** Credential.

```
# Faber Agent
transcript_cred_offer_json = await anoncreds.issuer_create_credential_offer(faber_wallet, faber_transcript_cred_def
```

Note: All messages sent between actors are encrypted using `Authenticated-encryption` scheme.

The value of this **Transcript** Credential is that it is provably issued by **Faber College**.

Alice wants to see the attributes that the **Transcript** Credential contains. These attributes are known because a Credential Schema for **Transcript** has been written to the Ledger.

```
# Alice Agent
get_schema_request = await ledger.build_get_schema_request(alice_faber_did, transcript_cred_offer['schema_id'])
get_schema_response = await ledger.submit_request(pool_handle, get_schema_request)
transcript_schema = await ledger.parse_get_schema_response(get_schema_response)

print(transcript_schema['data'])
# Transcript Schema:
{
    'name': 'Transcript',
    'version': '1.2',
    'attr_names': ['first_name', 'last_name', 'degree', 'status', 'year', 'average', 'ssn']
}
```

However, the **Transcript** Credential has not been delivered to Alice yet in a usable form. Alice wants to use that Credential. To get it, Alice needs to request it, but first she must create a **Master Secret**.

Note: A Master Secret is an item of Private Data used by a Prover 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 Prover). A Master Secret should be known only to the Prover.

Alice creates Master Secret in her wallet.

```
# Alice Agent
alice_master_secret_id = await anoncreds.prover_create_master_secret(alice_wallet, None)
```

Alice also needs to get the Credential Definition corresponding to the `cred_def_id` in the **Transcript** Credential Offer.

```
# Alice Agent
get_cred_def_request = await ledger.build_get_cred_def_request(alice_faber_did, transcript_cred_offer['cred_def_id'
get_cred_def_response = await ledger.submit_request(pool_handle, get_cred_def_request)
faber_transcript_cred_def = await ledger.parse_get_cred_def_response(get_cred_def_response)
```

Now Alice has everything to create a Credential Request of the issuance of the **Faber Transcript** Credential.

```
# Alice Agent
  (transcript_cred_request_json, transcript_cred_request_metadata_json) = \
      await anoncreds.prover_create_credential_req(alice_wallet, alice_faber_did, transcript_cred_offer_json,
                                                   faber_transcript_cred_def, alice_master_secret_id)
```

Faber prepares both raw and encoded values for each attribute in the **Transcript** Credential Schema. **Faber** creates the **Transcript** Credential for Alice.

```
# Faber Agent
# note that encoding is not standardized by Indy except that 32-bit integers are encoded as themselves. IS-786
transcript_cred_values = json.dumps({
    "first_name": {"raw": "Alice", "encoded": "113948171645748869017221791627810333 5"},
    "last_name": {"raw": "Garcia", "encoded": "532164278024179012358790245678912345 2"},
    "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"}
})

transcript_cred_json, _, _ = \
    await anoncreds.issuer_create_credential(faber_wallet, transcript_cred_offer_json, transcript_cred_request_jsor
                                             transcript_cred_values, None, None)
```

Now the **Transcript** Credential has been issued. Alice stores it in her wallet.

```
# Alice Agent
await anoncreds.prover_store_credential(alice_wallet, None, transcript_cred_request_json, transcript_cred_request_m
                                        transcript_cred_json, faber_transcript_cred_def, None)
```

Alice has it in her possession, in much the same way that she would hold a physical transcript that had been mailed to her.

Apply for a Job

At some time in the future, Alice would like to work for the fictional company, Acme Corp. Normally she would browse to their website, where she would click on a hyperlink to apply for a job. Her browser would download a connection request in which her Indy app would open; this would trigger a prompt to Alice, asking her to accept the connection with Acme Corp. Because we're using an Indy-SDK, the process is different, but the steps are the same. The process of the connection establishment is the same as when Faber was accepting the Steward connection request.

After Alice had established connection with Acme, she got the **Job-Application** Proof Request. A proof request is a request made by the party who needs verifiable proof of having certain attributes and the solving of predicates that can be provided by other verified credentials.

In this case, Acme Corp is requesting that Alice provide a **Job Application**. The Job Application requires a name, degree, status, SSN and also the satisfaction of the condition about the average mark or grades.

In this case, **Job-Application** Proof Request looks like:

```
# Acme Agent
job_application_proof_request_json = json.dumps({
    'nonce': '1432422343242122312411212',
    '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': faber_transcript_cred_def_id}]
        },
        'attr4_referent': {
            'name': 'status',
            'restrictions': [{'cred_def_id': faber_transcript_cred_def_id}]
        },
        'attr5_referent': {
            'name': 'ssn',
            'restrictions': [{'cred_def_id': faber_transcript_cred_def_id}]
        },
        'attr6_referent': {
            'name': 'phone_number'
```

```
            }
        },
        'requested_predicates': {
            'predicate1_referent': {
                'name': 'average',
                'p_type': '>=',
                'p_value': 4,
                'restrictions': [{'cred_def_id': faber_transcript_cred_def_id}]
            }
        }
    })
```

Notice that some attributes are verifiable and some are not.

The proof request says that SSN, degree, and graduation status in the Credential must be formally asserted by an issuer and schema_key. Notice also that the first_name, last_name and phone_number are not required to be verifiable. By not tagging these credentials with a verifiable status, Acme's credential request is saying it will accept Alice's own credential about her names and phone numbers.

To show Credentials that Alice can use for the creating of Proof for the **Job-Application** Proof Request Alice calls `anoncreds.prover_get_credentials_for_proof_req` .

```
    # Alice Agent
      creds_for_job_application_proof_request = json.loads(
          await anoncreds.prover_get_credentials_for_proof_req(alice_wallet, job_application_proof_request_json))
```

Alice has only one credential that meets proof the requirements for this **Job Application**.

```
    # Alice Agent
    {
      'referent': 'Transcript Credential Referent',
      'attrs': {
          'first_name': 'Alice',
          'last_name': 'Garcia',
          'status': 'graduated',
          'degree': 'Bachelor of Science, Marketing',
          'ssn': '123-45-6789',
          'year': '2015',
          'average': '5'
      },
      'schema_id': job_certificate_schema_id,
      'cred_def_id': faber_transcript_cred_def_id,
      'rev_reg_id': None,
      'cred_rev_id': None
    }
```

Now Alice can divide these attributes into the three groups:

1. attributes values of which will be revealed
2. attributes values of which will be unrevealed
3. attributes for which creating of verifiable proof is not required

For the **Job-Application** Proof Request Alice divided the attributes as follows:

```
    # Alice Agent
      job_application_requested_creds_json = 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']}}
    })
```

In addition, Alice must get the Credential Schema and corresponding Credential Definition for each used Credential, the same way, as on the step used to in the creation of the Credential Request.

Now Alice has everything to create the Proof for **Acme Job-Application** Proof Request.

```
# Alice Agent
apply_job_proof_json = \
      await anoncreds.prover_create_proof(alice_wallet, job_application_proof_request_json, job_application_request
                                          alice_master_secret_id, schemas_json, cred_defs_json, revoc_states_json)
```

When **Acme** inspects the received Proof he will see following structure:

```
# Acme Agent
{
    'requested_proof': {
        'revealed_attrs': {
            'attr4_referent': {'sub_proof_index': 0, 'raw':'graduated', 'encoded':'2213454313412354'},
            'attr5_referent': ['sub_proof_index': 0, 'raw':'123-45-6789', 'encoded':'3124141231422543541'},
            'attr3_referent': ['sub_proof_index': 0, 'raw':'Bachelor of Science, Marketing',
'encoded':'12434523576212321'}
        },
        'self_attested_attrs': {
            'attr1_referent': 'Alice',
            'attr2_referent': 'Garcia',
            'attr6_referent': '123-45-6789'
        },
        'unrevealed_attrs': {},
        'predicates': {
            'predicate1_referent': {'sub_proof_index': 0}
        }
    },
    'proof' : [] # Validity Proof that Acme can check
    'identifiers' : [ # Identifiers of credentials were used for Proof building
        {
          'schema_id': job_certificate_schema_id,
          'cred_def_id': faber_transcript_cred_def_id,
          'rev_reg_id': None,
          'timestamp': None
        }
    }
}
```

Acme got all the requested attributes. Now **Acme** wants to check the Validity Proof. To do it **Acme** first must get every Credential Schema and corresponding Credential Definition for each identifier presented in the Proof, the same way that Alice did it. Now **Acme** has everything to check **Job-Application** Proof from Alice.

```
# Acme Agent
assert await anoncreds.verifier_verify_proof(job_application_proof_request_json, apply_job_proof_json,
                                             schemas_json, cred_defs_json, revoc_ref_defs_json, revoc_regs_json)
```

Here, we'll assume the application is accepted and Alice ends up getting the job. **Acme** creates new Credential Offer for Alice.

```
# Acme Agent
job_certificate_cred_offer_json = await anoncreds.issuer_create_credential_offer(acme_wallet, acme_job_certificate_
```

When Alice inspects her connection with Acme, she sees that a new Credential Offer is available.

Apply for a Loan

Now that Alice has a job, she'd like to apply for a loan. That will require a proof of employment. She can get this from the **Job-Certificate** credential offered by Acme. Alice goes through a familiar sequence of interactions.

1. First she creates a Credential Request.

```
# Alice Agent
  (job_certificate_cred_request_json, job_certificate_cred_request_metadata_json) = \
      await anoncreds.prover_create_credential_req(alice_wallet, alice_acme_did, job_certificate_cred_offer_json,
                                                   acme_job_certificate_cred_def, alice_master_secret_id)
```

2. Acme issues a **Job-Certificate** Credential for Alice.

```
# Acme Agent
alice_job_certificate_cred_values_json = json.dumps({
    "first_name": {"raw": "Alice", "encoded": "245712572474217942457235975012103335"},
    "last_name": {"raw": "Garcia", "encoded": "312643218496194691632153761283356127"},
    "employee_status": {"raw": "Permanent", "encoded": "21431354254251431123213143 21"},
    "salary": {"raw": "2400", "encoded": "2400"},
    "experience": {"raw": "10", "encoded": "10"}
})
job_certificate_cred_json, _, _ = \
    await anoncreds.issuer_create_credential(acme_wallet, job_certificate_cred_offer_json,job_certificate_cred_reque
                                             alice_job_certificate_cred_values_json, None, None)
```

Now the **Job-Certificate** Credential has been issued and Alice now has it in her possession. Alice stores **Job-Certificate** Credential in her wallet.

```
# Alice Agent
await anoncreds.prover_store_credential(alice_wallet, None, job_certificate_cred_request_json, job_certificate_crec
                                        job_certificate_cred_request_json, acme_job_certificate_cred_def_json, None
```

She can use it when she applies for her loan, in much the same way that she used her transcript when applying for a job.

There is a disadvantage in this approach to data sharing though, — it may disclose more data than what is strictly necessary. If all Alice needs to do is provide proof of employment, this can be done with an anonymous credential instead. Anonymous credentials may prove certain predicates without disclosing actual values (e.g., Alice is employed full-time, with a salary greater than X, along with her hire date, but her actually salary remains hidden). A compound proof can be created, drawing from credentials from both Faber College and Acme Corp, that discloses only what is necessary.

Alice now establishes connection with Thrift Bank.

Alice gets a **Loan-Application-Basic** Proof Request from Thrift Bank that looks like:

```
# Thrift Agent
apply_loan_proof_request_json = json.dumps({
    'nonce': '123432421212',
    'name': 'Loan-Application-Basic',
    'version': '0.1',
    'requested_attributes': {
        'attr1_referent': {
            'name': 'employee_status',
            'restrictions': [{'cred_def_id': acme_job_certificate_cred_def_id}]
        }
```

```
        },
        'requested_predicates': {
            'predicate1_referent': {
                'name': 'salary',
                'p_type': '>=',
                'p_value': 2000,
                'restrictions': [{'cred_def_id': acme_job_certificate_cred_def_id}]
            },
            'predicate2_referent': {
                'name': 'experience',
                'p_type': '>=',
                'p_value': 1,
                'restrictions': [{'cred_def_id': acme_job_certificate_cred_def_id}]
            }
        }
    })
```

Alice has only one credential that meets the proof requirements for this **Loan-Application-Basic** Proof Request.

```
    # Alice Agent
    {
        'referent': 'Job-Certificate Credential Referent',
        'revoc_reg_seq_no': None,
        'schema_id': job_certificate_schema_id,
        'cred_def_id': acme_job_certificate_cred_def_id,
        'attrs': {
            'employee_status': 'Permanent',
            'last_name': 'Garcia',
            'experience': '10',
            'first_name': 'Alice',
             'salary': '2400'
        }
    }
```

For the **Loan-Application-Basic** Proof Request Alice divided attributes as follows:

```
    # Alice Agent
    apply_loan_requested_creds_json = json.dumps({
        'self_attested_attributes': {},
        'requested_attributes': {
            'attr1_referent': {'cred_id': cred_for_attr1['referent'], 'revealed': True}
        },
        'requested_predicates': {
            'predicate1_referent': {'cred_id': cred_for_predicate1['referent']},
            'predicate2_referent': {'cred_id': cred_for_predicate2['referent']}
        }
    })
```

Alice creates the Proof for the **Loan-Application-Basic** Proof Request.

```
    # Alice Agent
    alice_apply_loan_proof_json = \
        await anoncreds.prover_create_proof(alice_wallet, authdecrypted_apply_loan_proof_request_json, apply_loan_reque
                                            alice_master_secret_id, schemas_json, cred_defs_json, revoc_states_json)
```

Alice sends just the **Loan-Application-Basic** proof to the bank. This allows her to minimize the PII (personally identifiable information) that she has to share when all she's trying to do right now is prove basic eligibility.

When **Thrift** inspects the received Proof he will see following structure:

```
    # Thrift Agent
    {
```

```
        'requested_proof': {
            'revealed_attributess': {
                'attr1_referent': {'sub_proof_index': 0, 'raw':'Permanent',
'encoded':'21431354254251431123213143 21'},
            },
            'self_attested_attrs': {},
            'unrevealed_attrs': {},
            'predicates': {
                'predicate1_referent': {'sub_proof_index': 0},
                'predicate2_referent': {'sub_proof_index': 0}
            }
        },
        'proof' : [] # Validity Proof that Thrift can check
        'identifiers' : [ # Identifiers of credentials were used for Proof building
            'schema_id': job_certificate_schema_id,
            'cred_def_id': acme_job_certificate_cred_def_id,
            'revoc_reg_seq_no': None,
            'timestamp': None
        ]
    }
```

Thrift Bank successfully verified the **Loan-Application-Basic** Proof from Alice.

```
    # Thrift Agent
    assert await anoncreds.verifier_verify_proof(apply_loan_proof_request_json, alice_apply_loan_proof_json,
                                                 schemas_json, cred_defs_json, revoc_defs_json, revoc_regs_json)
```

Thrift Bank sends the second Proof Request where Alice needs to share her personal information with the bank.

```
    # Thrift Agent
    apply_loan_kyc_proof_request_json = json.dumps({
        'nonce': '123432421212',
        'name': 'Loan-Application-KYC',
        'version': '0.1',
        'requested_attributes': {
            'attr1_referent': {'name': 'first_name'},
            'attr2_referent': {'name': 'last_name'},
            'attr3_referent': {'name': 'ssn'}
        },
        'requested_predicates': {}
    })
```

Alice has two credentials that meets the proof requirements for this **Loan-Application-KYC** Proof Request.

```
    # Alice Agent
    {
      'referent': 'Transcript Credential Referent',
      'schema_id': transcript_schema_id,
      'cred_def_id': faber_transcript_cred_def_id,
      'attrs': {
          'first_name': 'Alice',
          'last_name': 'Garcia',
          'status': 'graduated',
          'degree': 'Bachelor of Science, Marketing',
          'ssn': '123-45-6789',
          'year': '2015',
          'average': '5'
      },
      'rev_reg_id': None,
      'cred_rev_id': None
    },
    {
        'referent': 'Job-Certificate Credential Referent',
        'schema_key': job_certificate_schema_id,
        'cred_def_id': acme_job_certificate_cred_def_id,
```

```
        'attrs': {
            'employee_status': 'Permanent',
            'last_name': 'Garcia',
            'experience': '10',
            'first_name': 'Alice',
            'salary': '2400'
        },
        'rev_reg_id': None,
        'revoc_reg_seq_no': None
    }
```

For the **Loan-Application-KYC** Proof Request Alice divided attributes as follows:

```
    # Alice Agent
    apply_loan_kyc_requested_creds_json = json.dumps({
        'self_attested_attributes': {},
        'requested_attributes': {
            'attr1_referent': {'cred_id': cred_for_attr1['referent'], 'revealed': True},
            'attr2_referent': {'cred_id': cred_for_attr2['referent'], 'revealed': True},
            'attr3_referent': {'cred_id': cred_for_attr3['referent'], 'revealed': True}
        },
        'requested_predicates': {}
    })
```

Alice creates the Proof for **Loan-Application-KYC** Proof Request.

```
    # Alice Agent
    alice_apply_loan_kyc_proof_json = \
        await anoncreds.prover_create_proof(alice_wallet, apply_loan_kyc_proof_request_json, apply_loan_kyc_requested_c
                                            alice_master_secret_id, schemas_json, cred_defs_json, revoc_states_json)
```

When **Thrift** inspects the received Proof he will see following structure:

```
    # Thrift Agent
    {
        'requested_proof': {
            'revealed_attributes': {
                'attr1_referent': {'sub_proof_index': 0, 'raw':'123-45-6789', 'encoded':'3124141231422543541'},
                'attr1_referent': {'sub_proof_index': 1, 'raw':'Alice',
'encoded':'24571257247421794245723597501210333 5'},
                'attr1_referent': {'sub_proof_index': 1, 'raw':'Garcia',
'encoded':'31264321849619469163215376128335612 7'},
            },
            'self_attested_attrs': {},
            'unrevealed_attrs': {},
            'predicates': {}
        },
        'proof' : [] # Validity Proof that Thrift can check
        'identifiers' : [ # Identifiers of credentials were used for Proof building
            {
              'schema_id': transcript_schema_id,
              'cred_def_id': faber_transcript_cred_def_id,
              'rev_reg_id': None,
              'timestamp': None
            },
            {
              'schema_key': job_certificate_schema_id,
              'cred_def_id': acme_job_certificate_cred_def_id,
              'rev_reg_id': None,
              'timestamp': None
            }
        ]
    }
```

Thrift Bank has successfully validated the **Loan-Application-KYC** Proof from Alice.

```
    # Thrift Agent
    assert await anoncreds.verifier_verify_proof(apply_loan_kyc_proof_request_json, alice_apply_loan_kyc_proof_json,
                                                 schemas_json, cred_defs_json, revoc_defs_json, revoc_regs_json)
```

Both of Alice's Proofs have been successfully verified and she got loan from **Thrift Bank**.

Explore the Code

Now that you've had a chance to see how the Libindy implementation works from the outside, perhaps you'd like to see how it works underneath, from code? If so, please run Simulating Getting Started in the Jupiter. You may need to be signed into GitHub to view this link. Also you can find the source code here

If demo gives an error when executing check Trouble Shooting Guide.