User manual

Using the Hyper42 Self-Sovereign Identity service for handling authorizations

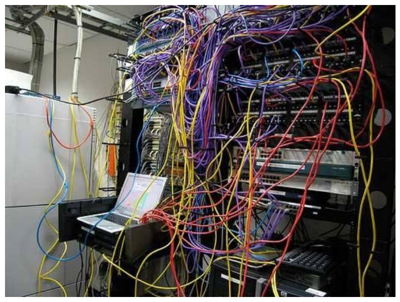
Welcome Momentum user! If you want to provide **authorizations** (**mandates**) for your **Self-Sovereign Identity** solution, then look no further!

Goal

In this guide, we will teach you how to connect your **agents** to ours running on the **Sovrin testnet**, powered by **Hyperledger Aries** and **Indy**. When this has been established, you will be able to give **authority** to an **identity** in order to perform actions on your behalf, like **authorizing** an **employee** to act on behalf of a **company**. This is also the use case which is described in this manual. Keep in mind that the employee here can also be called the **representative** requesting credentials, and the company is the **dependant** or **delegator** approving the requested credentials.

SSI gives the power of sharing your credentials back to where it belongs: The holders of the credentials, like YOU! As a person, as the owner of your own identity!

A bit overwhelmed? Understandable, this is a lot to take in. But no worries, we've got you covered! Within this guide, we will briefly explain the basics of SSI and its agents, which you can of course skip if you know the drill, as well as a complete step-by-step tutorial in order to connect to our service. Easy to connect and easy to maintain!



Have some questions?
Want to discuss some alterations in order to fit your needs?
We are happy to help!
Please reach out to the Hyper42 team :)
View the contact info at the end of this document.

Version 1.0

VISMA

Table of Contents

Background information

A quick overview of SSI basics

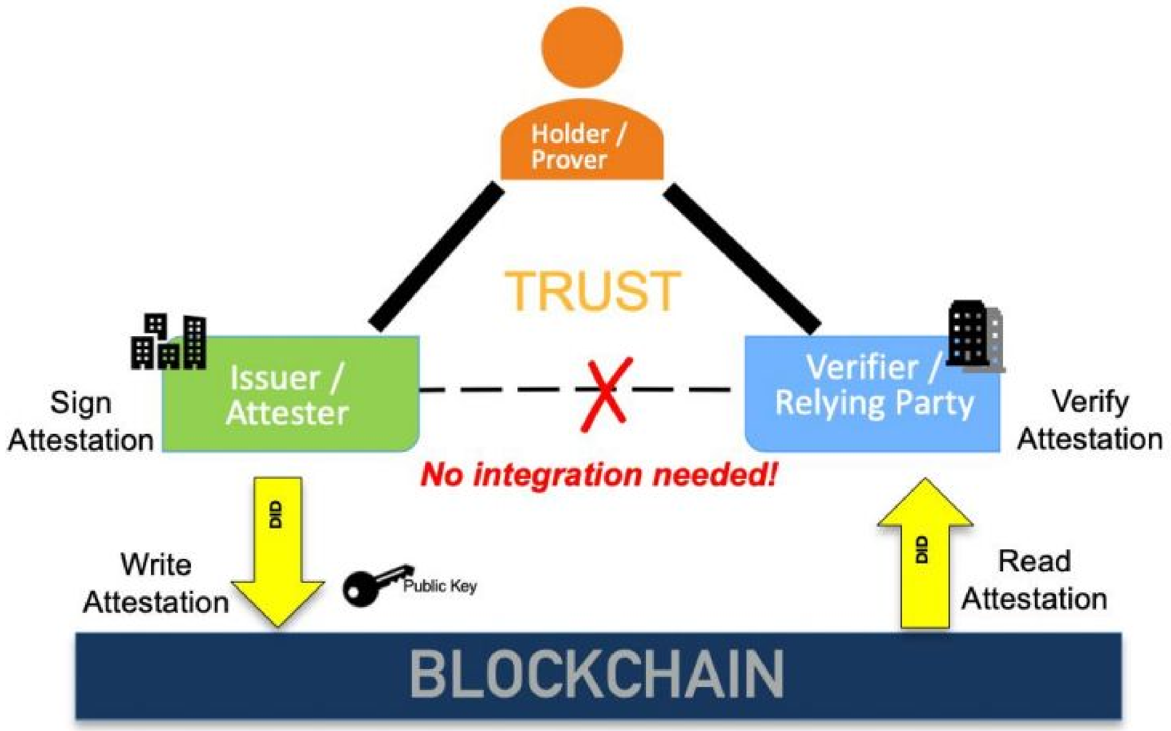
SSI has a 3 key roles:

Holder/Prover: Owner of the identity, for instance: you.

Issuer/Attester: A person or party who issues a credential to the holder, like a university issuing a diploma.

Verifier/Relying Party: A person, party or bot who needs your credentials for verification, for instance: a grocery store needs your age, so you can buy yourself a beer!

Below is a figure that shows the basic concept of SSI. SSI makes sure the verifier doesn't need integration with the issuer and can trust credentials stored in the prover's wallet which are signed with public decentralized identifiers (DID's) of the issuer. The public identifiers can be found on the blockchain from which the verifier can check whether the presented credentials of the holder are indeed valid.



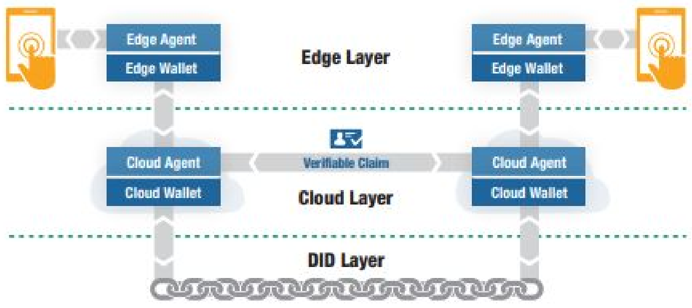
These roles are also functions the agents on the SSI-platform can perform (more on these later). There is an issuer function to issue signed attributes. A present function for a holder to present proof to a verifier. And lastly a verifier function to request and verify the presentation. Every actor (issuer/verifier/holder) has access to all of these functions and often uses all of them.

For instance, when a police officer asks for your driving license attribute as a holder, you want to use the verifier function to get proof that the police officer is indeed a police officer. When this checks out, the holder will use the present function to prove he/she has a driving license.

A verifier might also issue credentials themselves. For instance, a company issuing employer credentials required to enter the office. In this case, the company is both a verifier and an issuer. Even end-users might use the issuer function to issue a certain credential. Attestations are much broader than just information on your passport issued by governments/IDIN. It might even be an attestation you give to your neighbour which can be used to open the smart lock on your house.

Using endpoints with Agents

In this digital age, most digital identities run on smartphones. Smartphones don't have endpoints and therefore can't effectively receive messages. To create endpoints for communication, Sovrin and most other SSI solutions use agents. In Sovrin each DID has a corresponding private agent with its own pseudonymous network address from which the identity owner can exchange verifiable claims and any other data with another identity owner over an encrypted private channel as shown below:



Private agents can operate on edge devices (mobile phones, tablets, laptops, etc.), in the cloud, or both.

Note, an agent does not have its own DID. DIDs are orthogonal to Agents. Agents can be identified by a local name, or by the public key used in a particular relationship. Each DID/Agent combination requires a separate key. An endpoint must be unique per relationship, so an agent must support multiple endpoints. Agents are authorized for certain types of activity for each relationship. This allows the Agent owner to use different agents in different contexts.

The following description of agents within SSI, is part of Hyperledger Indy's documentation:

"When we use the term "agent" in the SSI community, we more properly mean "an agent of self-sovereign identity." This means something more specific than just a "user agent" or a "software agent." Such an agent has three defining characteristics:

1. It acts as a fiduciary on behalf of a single identity owner (or, for agents of things like IoT devices, pets, and similar things, a single *controller*).
2. It holds cryptographic keys that uniquely embody its delegated authorization.
3. It interacts using interoperable DID Comm protocols.

These characteristics don't tie an agent to Indy or to a blockchain that has Indy DNA. It is possible to implement agents without any use of Indy, and some efforts outside the Indy dev community are quite active."

In short, agents handle the following for you

- Communication between other agents (representative, dependent).
- Creating DIDs with other parties.
- Managing a wallet.
- Communication with the SSI network .
- Exchanging message using DIDs (DIDCOMM)
- Blockchain agnostic because of a modular setup. (Eg ULA plugin)

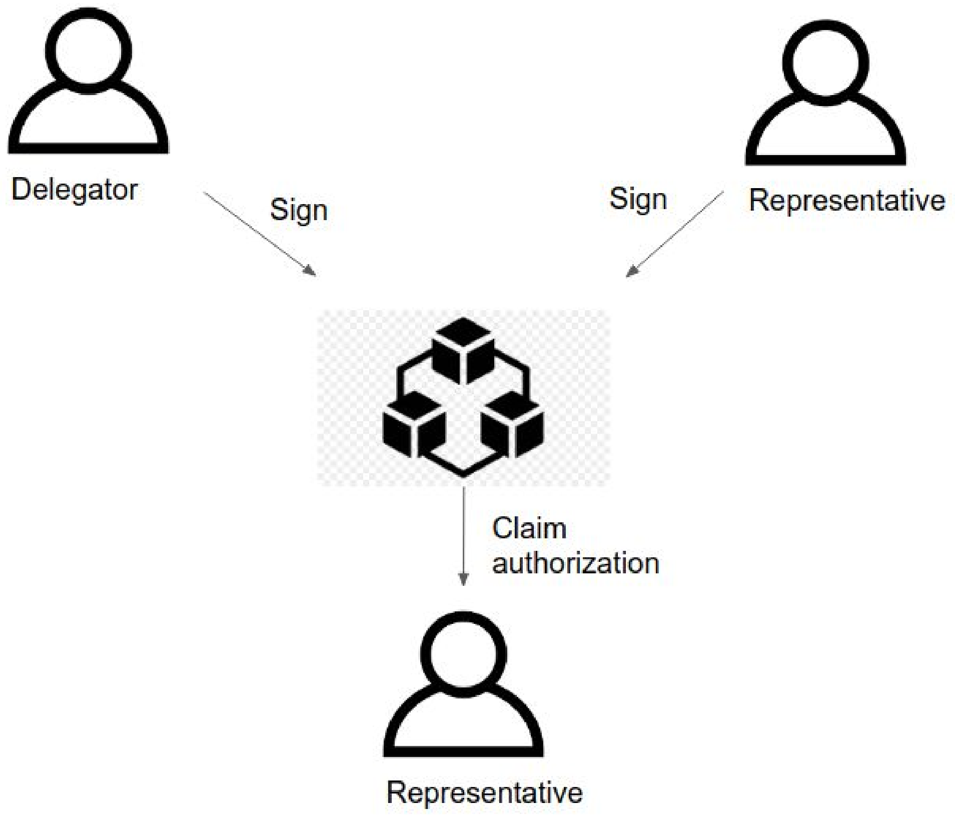
We based our service on the **Aries Cloudagent**. So if you are unsure which one to use, you can use this one. Another compatible agent is the **Trinsic Wallet,** which is an app for **IOS** and **Android**. When choosing an agent for our service, it is important to be able to accept the invite to our service, but also send messages to our service. More information about this can be found in the **requisites** within the tutorial in the second part of this manual.

Authorization module

SSI is a concept that is slowly being embraced by government bodies to become a future standard and it has proven itself in basic pilots where a holder presents a few verifiable credentials about him/herself issued by the government or bank. A missing link in SSI is however authorization. In many cases, a holder wants to have someone else act on their behalf. The application of authorizations is very broad. It could be guardianship where someone with mental decline appoints a guardian to make medical decisions when he/she is no longer able to. It might be a holder authorizing a family member or accountant to fill in tax forms. Even a shared bank account is a form of authorization. One can also authorize someone to pick up a package on their behalf or authorize another person to vote on their behalf. The use cases are endless and at this moment SSI has no answer to this issue. In its core concept, SSI is built around a holder presenting their own credentials and managing their own credentials.

Current solutions are typically authorization forms either in paper or PDF which require a physical old-fashioned signature that can easily be faked. Digital solutions are sometimes there but are typically not user-friendly resulting in people sharing their passwords and store personal identifiers in centralized systems to keep track of who is authorized for what.

Our idea is to provide authorizations with an authorization attribute which can be requested by the dependent for an authorized representative and can be used by the authorized representative to prove at a verifier he/she is authorized to act on the dependent's behalf. This way no centrally controlled identifiers need to be stored at a central party. A conceptual overview of how this works is shown below:



Connecting to our service

Requisites for connecting

- You will need **two** agents running locally, which are compatible with **libindy** and are able to send messages (in this case: the employee which requests authorizations and the company which is accepting authorizations). Both will communicate via our own cloud agent (this is a running **Aries Cloudagent** which can be accessed through the internet).
  Technically, both local running agents can be any agent who corresponds to the DID Comm protocols, but for now, we are focussed on getting identities working on **Sovrin**.
  - The **employee** (or **representative**) will most likely be using an **edge agent**, because he/she does not need to be connected at all times. Within this tutorial, we will use an app for this called **Trinsic Wallet** which you can download in the Google Play Store or Apple App Store (but you can also use another agent, like we are doing for the company).
    This app will allow the employee to accept the invitation to connect to our service by scanning a **QR code**, as well as request mandates from a specific company by sending a message to our service. This is something not all apps can do (like Connect.Me from Evernym for instance, which you can connect with, but can't send messages from).
    Assigned **credentials** will be saved within the Trinsic Wallet app.
  - The **company** (or **dependant**, **delegator**) will use the **Aries Cloudagent** in this guide (but you can ofcourse also use an app for this, like we are doing for the employee). This will allow you to accept the invitation to connect to our service, request a **dependant-key** which is needed when employees request mandates, and send mandate approval response messages.
    Please view their Github repository for setup instructions. This will run in a Docker. You can follow this guide to play around with the agent locally.
    Credentials will be saved within the wallet of the agent.

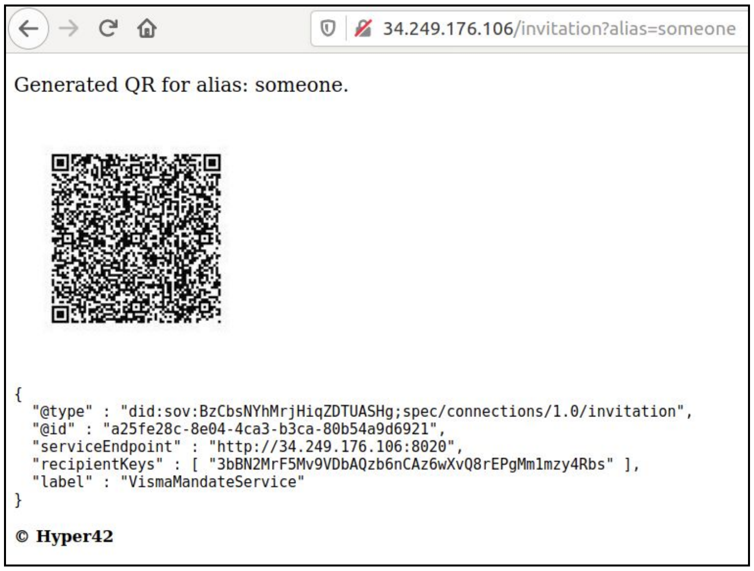
Optionally, you can also perform these calls via libindy yourself using a wrapper (Indy SDK), so you can make your own agent if you desire.

Step-by-step tutorial

1. Creating an invitation to communicate with our cloud agent

Before we can make a connection with your agents, they need to have received and accepted an invitation, created by our cloud agent. Be aware that you have to do this for all the agents you want to connect to our service.
The first step is to let our cloud agent create an invitation request. This step is really easy!
The service we offer is located at *http://34.249.176.106*, this is also the place where you've likely stumbled on the link to this manual. Well, you simply add /invitation?alias=<ALIAS> to our hostname and you've got yourself an invitation QR code page! You can replace <ALIAS>

with whatever you would like, but we advise you to use a name which is clear for you and unlikely to be picked by anyone else.
If visited correctly, you would see the following: A QR Code you can represent to your user for scanning, and some json code we will need in order to receive the invitation. We will need both in the next step.

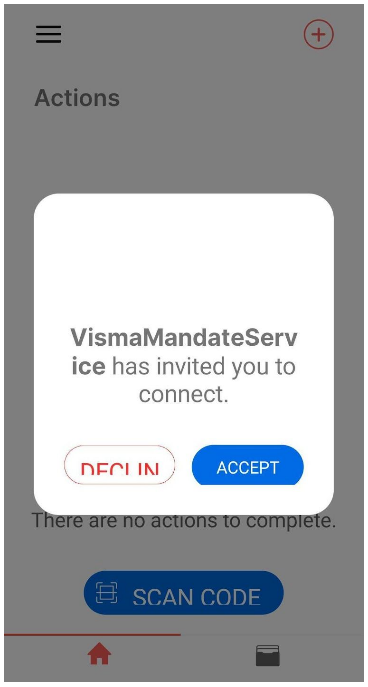


Generated QR for alias: someone.

```
{
  "@type" : "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/connections/1.0/invitation",
  "@id" : "a25fe28c-8e04-4ca3-b3ca-80b54a9d6921",
  "serviceEndpoint" : "http://34.249.176.106:8020",
  "recipientKeys" : [ "3bBN2MrF5Mv9VDbAQzb6nCAz6wXvQ8rEPgMm1mzy4Rbs" ],
  "label" : "VismaMandateService"
}
```

© **Hyper42**

2. Receiving and accepting the invitation for your agents



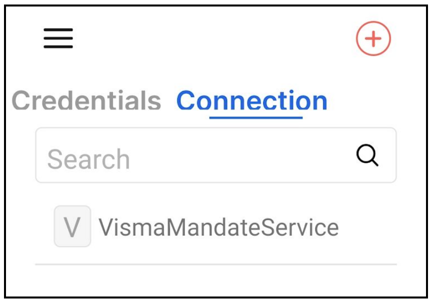
VismaMandateServ**ice** has invited you to connect.

There are no actions to complete.

Accepting the invitation as the representative/employee (Trinsic Wallet app)

The first thing a typical user, in this case the employee of the company, would do, is scan the QR-code seen above. For now we will scan this using the Trinsic Wallet. If you scan the QR-code without such an app, you may see the following message:
“You have received a connection invitation. To accept the invitation, paste it into your agent application.”
We advise you to select the sovrin staging network within Trinsic.
After setting up the app, scan the code you’ve got on the invitation page. Press accept to complete the invitation process between the employee and our mandate service.

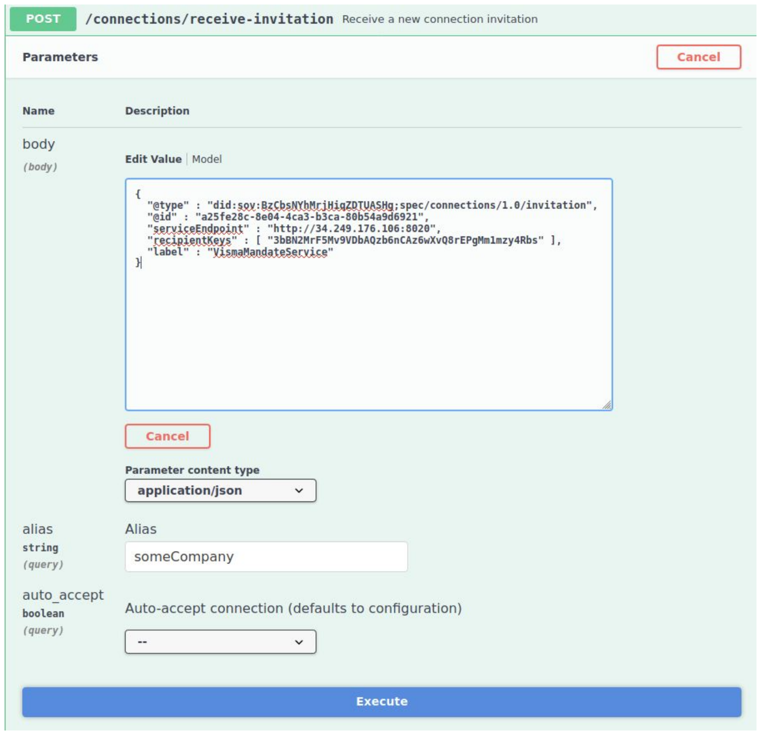
Now, you are able to see the connection in your wallet. This is also the place where you can send messages to the company via our service, but this will be covered in the next step. First, we also need to connect the company’s agent to our mandate service as well.



Accepting the invitation as the dependant/company (Aries Cloudagent)

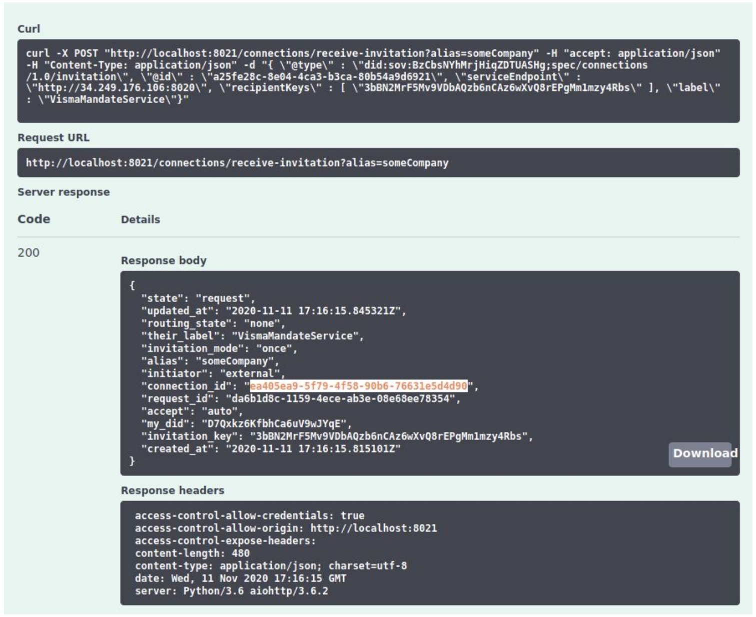
When running an agent using Aries Cloudagent, we need to first receive and accept the invitation in order to gather the correct **connection ID**. In order to receive the invitation, we need to copy the json invitation code returned from the (newly refreshed) QR-code page into the body of the post request to /connections/invitation within your local agent. If done correctly, we will get back the connection ID we need in order to accept the invitation.
We are using the OpenApi definitions in order to submit the requests.

Request: (note, the alias is optional)



```
{
  "@type" : "did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/connections/1.0/invitation",
  "@id" : "a25fe28c-8e04-4ca3-b3ca-80b54a9d6921",
  "serviceEndpoint" : "http://34.249.176.106:8020",
  "recipientKeys" : [ "3bBN2MrF5Mv9VDbAQzb6nCAz6wXvQ8rEPgMm1mzy4Rbs" ],
  "label" : "VismaMandateService"
}
```

Response:



```
curl -X POST "http://localhost:8021/connections/receive-invitation?alias=someCompany" -H "accept: application/json"
-H "Content-Type: application/json" -d "{ \"@type\" : \"did:sov:BzCbsNYhMrjHiqZDTUASHg;spec/connections
/1.0/invitation\", \"@id\" : \"a25fe28c-8e04-4ca3-b3ca-80b54a9d6921\", \"serviceEndpoint\" :
\"http://34.249.176.106:8020\", \"recipientKeys\" : [ \"3bBN2MrF5Mv9VDbAQzb6nCAz6wXvQ8rEPgMm1mzy4Rbs\" ], \"label\"
: \"VismaMandateService\"}"
```

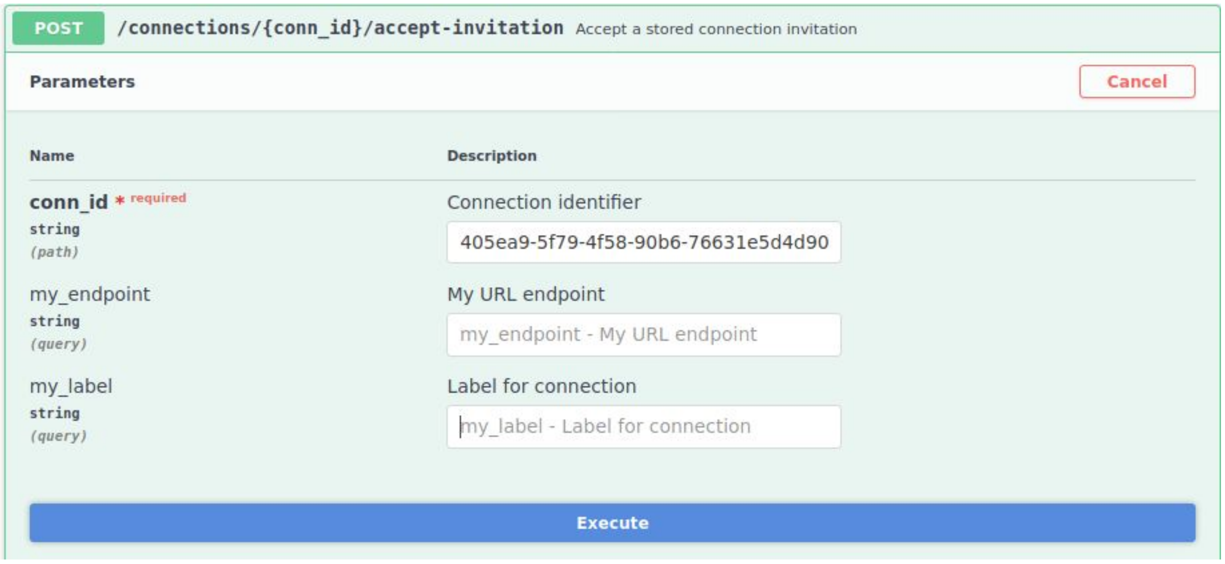
```
http://localhost:8021/connections/receive-invitation?alias=someCompany
```

```
{
  "state": "request",
  "updated_at": "2020-11-11 17:16:15.845321Z",
  "routing_state": "none",
  "their_label": "VismaMandateService",
  "invitation_mode": "once",
  "alias": "someCompany",
  "initiator": "external",
  "connection_id": "ea405ea9-5f79-4f58-90b6-76631e5d4d90",
  "request_id": "da6b1d8c-1159-4ece-ab3e-08e68ee78354",
  "accept": "auto",
  "my_did": "D7Qxkz6KfbhCa6uV9wJYqE",
  "invitation_key": "3bBN2MrF5Mv9VDbAQzb6nCAz6wXvQ8rEPgMm1mzy4Rbs",
  "created_at": "2020-11-11 17:16:15.815101Z"
}
```

```
access-control-allow-credentials: true
access-control-allow-origin: http://localhost:8021
access-control-expose-headers:
content-length: 480
content-type: application/json; charset=utf-8
date: Wed, 11 Nov 2020 17:16:15 GMT
server: Python/3.6 aiohttp/3.6.2
```

Now, we can accept the invitation using the connection ID we've got back. For this, we need to do a post request to /connections/<connectionID>/accept-invitation. We will replace <connectionID> with the ID we've got back in the previous call.

Request:

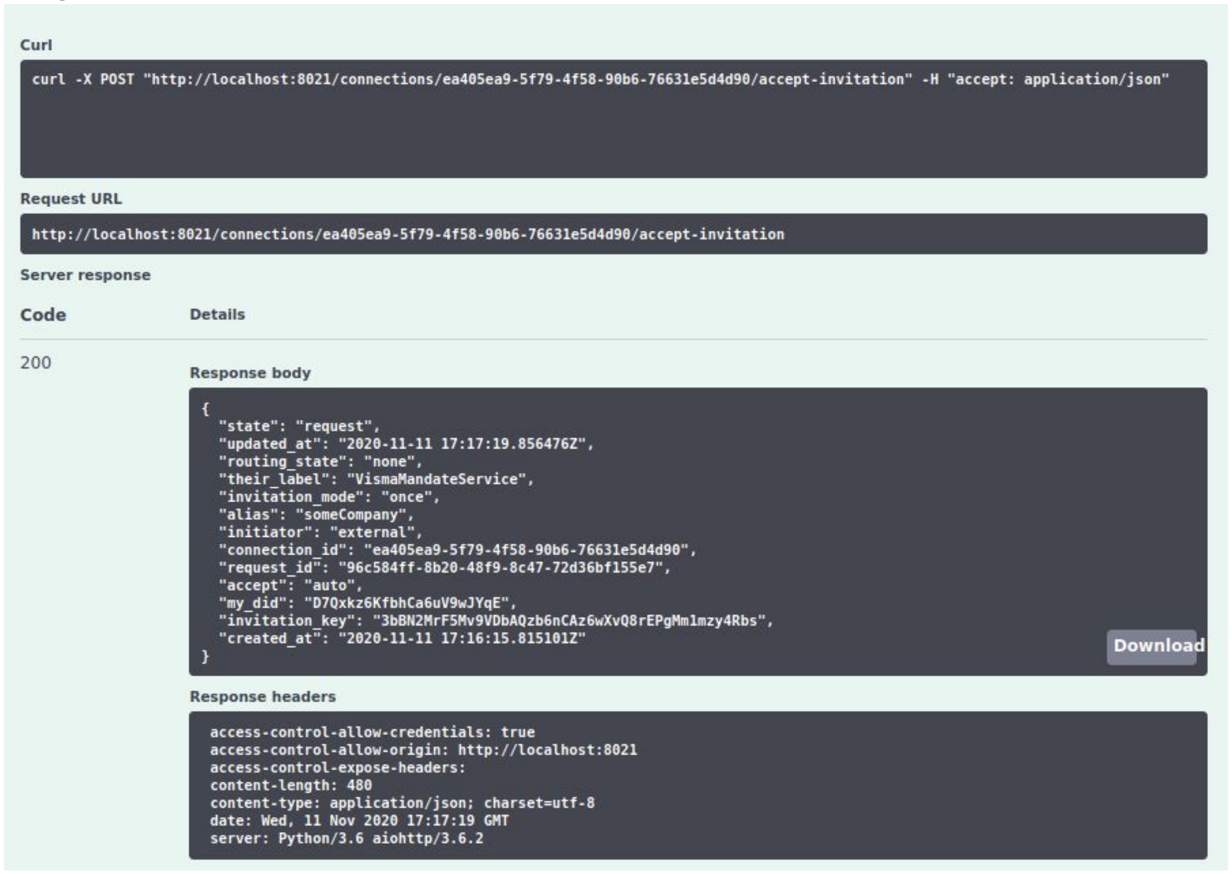


POST `/connections/{conn_id}/accept-invitation` Accept a stored connection invitation

Parameters

Name	Description
conn_id * required string *(path)*	Connection identifier 405ea9-5f79-4f58-90b6-76631e5d4d90
my_endpoint string *(query)*	My URL endpoint my_endpoint - My URL endpoint
my_label string *(query)*	Label for connection my_label - Label for connection

Response:



Curl

```
curl -X POST "http://localhost:8021/connections/ea405ea9-5f79-4f58-90b6-76631e5d4d90/accept-invitation" -H "accept: application/json"
```

Request URL

```
http://localhost:8021/connections/ea405ea9-5f79-4f58-90b6-76631e5d4d90/accept-invitation
```

Server response

Code	Details
200	Response body

```
{
  "state": "request",
  "updated_at": "2020-11-11 17:17:19.856476Z",
  "routing_state": "none",
  "their_label": "VismaMandateService",
  "invitation_mode": "once",
  "alias": "someCompany",
  "initiator": "external",
  "connection_id": "ea405ea9-5f79-4f58-90b6-76631e5d4d90",
  "request_id": "96c584ff-8b20-48f9-8c47-72d36bf155e7",
  "accept": "auto",
  "my_did": "D7Qxkz6KfbhCa6uV9wJYqE",
  "invitation_key": "3bBN2MrF5Mv9VDbAQzb6nCAz6wXvQ8rEPgMm1mzy4Rbs",
  "created_at": "2020-11-11 17:16:15.815101Z"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: http://localhost:8021
access-control-expose-headers:
content-length: 480
content-type: application/json; charset=utf-8
date: Wed, 11 Nov 2020 17:17:19 GMT
server: Python/3.6 aiohttp/3.6.2
```

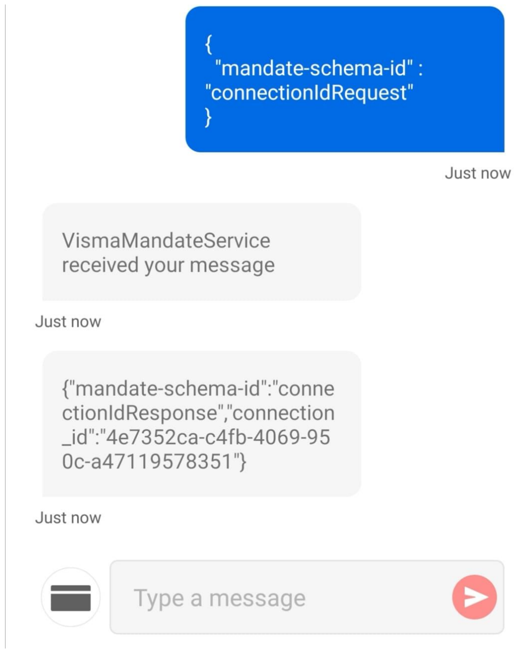
After accepting the invitations, the transactions are stored on the Sovrin Testnet. We now have a trusted connection with pairwise DIDs between our service and your agents!

Before we can request mandates however, as a company, you need to get a **dependant-key**. This is, for now, the same as the connectionID. You need to pass this dependant-key to the representatives who want to issue credentials on your behalf, as they need it within their mandate request (which is the next step). This way, we know for what dependant or **delegator** the representative wants to issue credentials for. In our case: the **dependant-key** is tied to a company, and representatives need this when issuing requests.

How to get the dependant-key?

When using the Aries Cloudagent, you do not need to do anything, as you've already got your connectionID. However, if you are a company who uses the Trinsic Wallet app, you do not know your connectionID. No worries! You can send us the following message and we will give you your connectionID (example screenshot is from the Trinsic app, where you cannot view your connectionID). This works for every user within our system.

```
{
  "mandate-schema-id" : "connectionIdRequest"
}
```



{
 "mandate-schema-id" :
"connectionIdRequest"
}

Just now

VismaMandateService
received your message

Just now

{"mandate-schema-id":"connectionIdResponse","connection_id":"4e7352ca-c4fb-4069-950c-a47119578351"}

Just now

If correct, you will get back the response with the connectionID/dependant-key.

3. Requesting the authorization (representative/employee, Trinsic Wallet)

This feature might not be ready yet, please consult the team if you are facing issues.

Types of credentials for our basic schema

We have created a basic schema for everyone to use. This schema holds many different credentials. **CUSTOM SCHEMA'S ARE POSSIBLE!**
Every **credential** or **authorization** the **representative** (in our case, the **employee**) can issue, has a unique key and value. These values are all Strings. We define these credentials in the following **key-value pairs**:

Key	Description
chamberofcommerce-reference	8-digit identification number for the Dutch Chamber of Commerce (Kamer van Koophandel, KvK)
spending-limit	The max amount to spend on something
enddate	The date when the authorization should end
issuer-id	The ID of the issuing party (representative?)
representative-	Representative, in our use case: the company
representative-age	Age of representative party
dependent-familyname	Family name of dependant, in our case: The employee
representative-givenname	Given name of the representative
dependent-id	The ID of the depending party, in our case: The company
dependent-givenname	The given name of the depending party
issuedate	The date of issuing the credential
authorization-accepted	Whether the authorization has been accepted
signed-by	Signed-by signature or name
timestamp	Timestamp of sending request
representative-familyname	Family name of representing party
dependent-age	Age of depending party
authorization-type	Type or authorization
proof	Given proof
startdate	The date when the authorization for said credential

	starts

Want something else? Please contact us and request a schema! We'll make it happen!

Now the **employee** and **company** have separate connections with our service, we can request **authorizations**. Because Indy does not really have an option for this, we will define the request within a **basicmessage**, which you can send within your **Trinsic** app (or optionally, with **Aries Cloudagent**).
We need the following things before the employee can make the request:

- The **mandate-schema-id**: Clarifies to our service what kind of message this is, in our case: "MandateRequest".
- The **requested-schema**: Schema ID which is defined on the ledger. This is for now just one schema for signing: "55MgvkQDB815vtbFky4kEt:2:mandate-schema:1.1"
- The **representative**: Can be any value, like a name, for clarification.
- The **dependant**: The dependant-key you received from the company in the last step. This can be a dummy value if there is no mandate-key feature ready yet.
- The **authorizations** (one or more) which each have the following fields:
  - The **key**: View the table above.
  - The **value**: view the table above.

The message the employee will send, will have the following json format:

```
{
  "mandate-schema-id" : "MandateRequest",
  "representative": "John Doe",
  "dependant": "adlsjkfqljf-34jlj3-zckjvajdf",
  "authorizations": [
    {
      "key": "chamberofcommerce-reference",
      "value": "12312312"
    }
  ]
}
```

<TODO: Add sending the json request within the Trinsic Wallet app for the employee to our service>

If validated correctly, our service will send the message over to the company for approval.

4. Receiving and accepting the authorization request (dependant/company, Aries Cloudagent)

This feature might not be ready yet, please consult the team if you are facing issues.

The company now has a pending authorization request. The company will get this request within a basic message sent by our service if the request is validated successfully.
<TODO: Add receiving the json mandate request within the Aries Cloudagent for the company<

We received the authorization request. As you can see, it now has a **mandate-key**. We will need this mandate-key in order to approve the request.

Now the company can accept the request by sending a message to our service. For this, we need the following fields:

- The **mandate-schema-id**: Clarifies to our service what kind of message this is, in our case: “MandateApprovalResponse”.
- The **status**: For now, can only be “Approved”..
- The **authorizations** (one or more) which each have the following fields:
  - The **key**: Like discussed in the last step.
  - The **value**: Like discussed in the last step.
  - **Signature**: A signature to verify the approval of the specific authorization. For now, this can be any String.
- **Mandate-key:** Key between our service and the company. This has been sent with the authorization request. This is needed to determine who has sent the request.

The message the company will send, will have the following json format:

```
{
  "mandate-schema-id" : "MandateApprovalResponse",
  "status": "Approved",
  "signature": "ajkl;dfalkjdrfqw4opiru5cvnasdkl;jf;lqwerj",
  "authorizations": [
    {
      "key": "key1",
      "value": "value1",
      "signature": "130489dslkjfn123"
    }
  ],
  "mandate-key": "ffefc638-7842-4e64-8346-c90128bfe862"
}
```

<TODO: Add sending the json request within the Aries Cloudagent of the company to our service>

5. Further steps and contact information

After the mandates have been approved successfully, our service will issue the credentials for the employee.
<TODO: Add showing credentials of employee in Trinsic Wallet>

For now, revoking authorizations by the company is not yet implemented, but this is on the roadmap. Ofcourse, the employee can always remove credentials, as they are part of his own identity.

If you want some adjustments in order to connect our service to your solution or are in need of help, please reach out to us!

Contact us

Normally, you would be able to visit us by walking by our table, unfortunately, this is not possible now.

However, we still would very much like to get in touch with you. You can find us in the self-sovereign-identity Discord channel, or you can contact us via the contact details below:

Name	**Role**	**Discord**	**Phone**
Ralph Verhelst	Captain	Ralph Verhelst - Hyper42#6566	+31628597305
Raman Hossain	Connect aid	Raman100 - Hyper42#9143	-
Rogier Morsink	Developer	Rogier - Hyper42#0470	-
Ragesh Shunmugam	Developer	Ragesh Sharma - Hyper42#3327	-
Kevin Kerkhoven	Developer	Kevin - Hyper42#6147	-
Amit Mohabir	Tester	Amit - Hyper42#4081	-
Tim Janssen	Blockchain overlord	Tim Janssen#8682	+31625734977
Marcel Raap	Blockchain business overlord	MarcelRaap#5644	+31630724939

You can also reach us by email: firstname.lastname@visma.com