MOSIP

MODULAR OPEN SOURCE IDENTITY PLATFORM

A country's identification system is the backbone for effective delivery of public and private services. Governments are exploring the development of multipurpose foundational ID systems, in which individuals receive a unique identifier from the government that they can use for identity assertion and verification. The foundational ID can then be used to access a wide variety of government and private services.

As countries consider how best to build foundational ID systems, they face several policy and technological choices. Once these choices are made, countries often grapple with some common implementation challenges. These include ensuring uniqueness in the system, interoperability, privacy by design reaching scale, avoiding vendor lock-in and maintaining affordability.

MOSIP is a modular and open source identity platform that helps user organisations such as Governments implement a digital, foundational ID in a cost effective way, while embracing the best practices of scalability, security and privacy harnessing the power of open source.

Being modular in its architecture, MOSIP provides lot of flexibility in how they implement and configure their foundation ID system. It is a unique, universal, and progressive digital identity system which is also an open source platform that nations can reuse freely and build their own identity systems.

Table of Contents:

Project: *Partner On-boarding & Credentials Generator*

Objectives:

1. Partner On-boarding
   a. Develop scripts to create certificates and onboard partners
2. Credentials generator
   a. An application which generates the credentials and uploads to IDA
   b. This application should generate unique persona with biometrics such as photo, finger prints, IRIS etc and demographic details such as name, gender, dob, address etc and convert to credential json format required for upload to IDA
   c. This application will make use of code available in Packet utility

Repository Links:
https://github.com/mosip/mosip-automation-tests/tree/main/mosip-packet-creator

https://github.com/mosip/mosip-automation-tests/tree/main/mosipTestDataProvider

This uses the Anguli application.

Tech Stack: Java, Spring Boot, REST API

IDA Data Generator:

Project Introduction:

Use case developers would be required to integrate with MOSIP Authentication and E-KYC features. For this they need to access the relevant APIs and should be able to configure the Partners for the same. The MOSIP Sandbox deployment would require a large number of VMs and complexes to setup and configure. We would like to develop a procedure to deploy only required modules and configure the same for integration and testing

Understanding:

Project Folder Structure:

`mosip-automation-tests/mosip-packet-creator/src/main/java/io/mosip/test/packetcreator/mosippacketcreator`

Controller files are in the `“controller”` directory in the above path and Service files are in the `“services”` directory in the above path.

The test data provider functions are in `mosipTestDataProvider` folder.

Work Done:

We added a new GetMapping `/persona/generateIdentityData` inside `PersonaController.java` This API creates test persona identity as per the given specification on this page.

This GetMapping calls the `generateIdentityData()` method in `packetSyncService.java` and returns its output.

The `packetSyncService.generateIdentityData()` method takes a resident’s JSON file as input and calls `CreatePersonaIDA.createIdentityIDA()` method which is defined at the following path under the `mosipTestDataProvider` folder,

`src/org/mosip/dataprovider/test/CreatePersonaIDA.java`

The `generateIdentityData()` method returns a new JSON object response with the fields:

`status:` `“SUCCESS”`
`response:` `generatedIdentity`

where, `generatedIdentity` is the JSON output of the `createIdentityIDA()` method.

The `createIdentityIDA()` method reads resident JSON data from Persona File which is given as the input and it creates a `ResidentModel` object with the help of the `ResidentModel.readPersona()` method. It then gets the latest IDSchema with the help of the several API calls. For these API calls to work, we had to update several `.properties` files with the updated authentication credentials in order to validate our requests.

It then creates a new `JSONObject identity` and performs the following for every item in `schemaList`:

- Add log statement for current `schemaItem`
- Skip `schemaItem` if it doesn't exist in `requiredAttributes`

- Skip `schemaItem` if it is present in `missingAttribute`
- Skip `document` and `biometric` type `schemaItem`
- Skip `IDSchemaVersion` `schemaItem`
- Skip `schemaItem` if it has no `type` attached to it
- Add `residentStatus` to the `identity` JSON
- Add `fullname` to the `identity` JSON
- Add `firstname`, `lastname` or `middlename` to `identity` JSON
- Add `dateOfBirth` or `dob` or `birthdate` to `identity` JSON
- Add `gender` to `identity` JSON
- Add `address` to `identity` JSON
- Add `email` to `identity` JSON
- Add `phone` to `identity` JSON
- Add `referenceIdentityNumber` to `identity` JSON
- Add location hierarchy related parameters to `identity` JSON which include,
    - `placeOfBirth`, `listCountry`, `region`, `province`, `city`, `postalCode`, `zone`
- If `resident` has no `UIN`, then set `UIN` to `null` in `identity` JSON, otherwise place the actual `UIN` in `identity`
- Add `biometrics` to `identity` JSON by first converting it to an XML file with the help of the `BiometricDataProvider.toCBEFF()` method. We then attach the filename of the XML file, i.e. `resident.getID() + "_biometrics.xml"` in `identity` JSON under the `individualBiometrics.value` key
- Add the introducer/guardian parameters to `identity` JSON, only applicable if the resident has a guardian, i.e. resident is minor.

Then it creates the final output JSON `response` as per the required format i.e. containing the following fields:

- id - dummy value
- registrationId - dummy value

- biometricReferenceId - dummy value
- identity - Same as `identity` JSON described above
- documents - `documents` JSON described below
- request - Contains the `identity` JSON as a key
- requesttime - dummy value
- version - dummy value

To add the documents, we are iterating through the documents that the resident has, i.e. `resident.getDocuments()` and converting each document `curDocument` to a JSON that looks as follows,

```
{
  “category”: curDocument.getDocCategoryName(),
  “value”: curDocument.getDocs()
}
```

Finally it returns the required `response` JSON object.

Team:

JASVIN JAMES (MT2021059)

HIMANK JAIN (MT2021054)

VISHAL RAI (MT2021153)

SHIVANI SHETH (MT2021126)

THAKUR DEVENDRASINGH (MT2021146)

BHARGAV MEHTA (MT2021029)