BROKERAGE AND MARKET PLATFORM FOR PERSONAL DATA

D3.2 Self-Sovereign Identity Solution Final Release

www.krakenh2020.eu

This project has received funding from the European Union's Horizon 2020 (H2020) research and innovation programme under the Grant Agreement no 871473

D3.2 Self-Sovereign Identity Solution Final Release

Grant agreement	871473
Work Package Leader	Atos
Author(s)	Angel Palomares (Atos)
Contributors	Angel Palomares Pérez (Atos), Juan Carlos Pérez Baún (Atos), Davide Porro (ICERT), Luca Boldrin (ICERT), Christof Rabensteiner (TUG).
Reviewer(s)	Christof Rabensteiner (TUG), Davide Zaccagnini (LYNKEUS)
Version	v1.0
Due Date	31/05/2022
Submission Date	02/06/2022
Dissemination Level	Public

Release History

Version	Date	Description	Released by
v0.1	13/01/2020	Initial version: ToC & Partnerts Assignation	Angel Palomares
v0.2	18/03/2022	Added diagrams and content on sections 2, 4 and 5. Contribution on section 6.	Angel Palomares (Atos), Christof Rabensteiner (TUG)
v0.3	20/04/2022	Updating introduction, KRER, DID registry and review.	Juan Carlos Pérez (Atos)
v0.4	29/04/2022	Completing sections 2 & 5. Added sections 3, 4.1.1 and 4.1.2	Angel Palomares (Atos), Davide Porro (ICERT)
v0.5	10/05/2022	Addressing comments after internal review	Angel Palomares (Atos), Davide Porro (ICERT)
v0.6	13/05/2022	Compiled version	Juan Carlos Pérez (Atos)
v0.7	23/05/2022	Addressing comments	Davide Porro (ICERT), Luca Boldrin (ICERT)
v0.8	25/05/2022	General review and including comments to stable version	Angel Palomares (Atos), Juan Carlos Pérez (Atos)
v0.9	27/05/2022	Addressing pending reviewers' comments	Davide Porro (ICERT), Luca Boldrin (ICERT)
V0.95	30/05/2022	Addressing reviewers' comments and compiling content	Davide Porro (ICERT), Angel Palomares (Atos), Juan Carlos Pérez (Atos)
v1.0	02/06/2022	Submitted version	Atos

Table of Contents

List of Figures

List of Acronyms

Acronym	Description
ACA-Py	Aries Cloud Agent Python
ACS	Access Control Service
AGID	Italian agency for digitalization
API	Application Programming Interface
AWS	Amazon Web Services
CA	Consortium Agreement
DBMS	DataBase Management system
DDBB	DataBase
DID	Decentralised Identifier
DIF	Decentralized Identity Foundation
DLT	Distributed Ledger Technology
DMS	Data Managed Services
EBSI	European Blockchain Service Infrastructure
eID	Electronic identity
eIDAS	electronic Identification, Authentication and trust Services
ESSIF	European Self-Sovereign Identity Framework
IPFS	Interplanetary File System
KMS	Key Manager Service
KRER	KRAKEN Revocation and Endorsement Registry
KTIR	KRAKEN Trusted Issuer Registry
KTSR	KRAKEN Trusted Scheme Registry
KWCT	KRAKEN Web Company Tool
LIM	Legal Identity Manager
PAdES	PDF Advanced Electronic Signatures
SP	Service Provider
SSI	Self-Sovereign Identity
VC	Verifiable Credential
VDR	Verifiable Data Registry
WP	Work Package

Executive Summary

Based on the architectural design developed in *T2.2 Existing Platforms Architecture Extension* and the specifications provided by task *T2.3 Self-sovereign Identity specifications*, WP3 provides the tools needed for facilitating personal data sharing between different actors and for access management leveraging the self-sovereign paradigm. This document contains the description of the building-blocks identified for creating the Self-Sovereign Identity (SSI) solution which is one of the three main pillars of the KRAKEN platform. As some parts of *D3.1 Self-Sovereign Identity Solution First Release* [4] (the first iteration of this deliverable) are still valid or only have slightly changed, this report includes information already reported in D3.1 (referenced properly) with the aim of providing a better reading experience. Therefore, this report details the final and comprehensive results of the following tasks:

- *T3.1 Verifiable Credential management tools*: Generation of derived verifiable credentials (VCs) from the data obtained after the authentication of an end user using eIDAS.
- *T3.2 Universal Ledger Resolver*: Implementation of components to manage public DIDs and to be aligned with the EBSI/ESSIF guidelines (Trusted Framework).
- *T3.3 Edge Tools for the Decentralised Identity Solution*: Development of two main components: Ledger uSelf Broker (for Service Providers) and Ledger uSelf Mobile App (for end users).
- *T3.4 SSI Wallet Management*: Developments performed for enabling a secure cloud-based storage system for backing up the personal data of end users.

The outcome of these activities is the final prototype of the SSI solution, one of the three main pillars of KRAKEN. This solution comprises updates of the elements included in the first version of the KRAKEN platform and new functionalities developed during the second iteration such as the KRAKEN Trusted Issuer Registry (KTIR), the KRAKEN Revocation & Endorsement Registry (KRER) and the DID resolution. These components will be integrated with the KRAKEN marketplace and used in the health and education pilots.

1 Introduction

1.1 Purpose of the document

This document provides a comprehensive report of the final outcomes obtained in the implementation of the WP3 tasks: T3.1, T3.2, T3.3 and T3.4. As some parts of *D3.1 Self-Sovereign Identity Solution First Release* [4], the first iteration of this deliverable, didn't change or slightly changed, this report includes information already reported in D3.1 (referenced properly) with the aim to provide a better reading experience.

Following the same criteria as in D3.1 the results included in this report are based on the input received from WP2, *D2.5 KRAKEN final technical design* [7], produced in the context of *T2.4 Self-sovereign Identity specifications*, and *D2.3 Final KRAKEN architecture* [8] generated in *T2.2* Existing platform architecture extension. The Self-Sovereign Identity solution produced in WP3 will be integrated with the KRAKEN marketplace in WP5 for its final validation.

1.2 Structure of the document

The structure of this report is detailed below, there is a chapter dedicated to each one of the WP3 tasks:

- Chapter 2: "SSI Solution overview" gives an introduction and a high-level view of the components implemented and explains how these components interact with each other.
- Chapter 3: "T3.1 Verifiable Credential management tools components" details the main outcomes and status of the developments performed under the umbrella of the Legal Identity Manager (LIM), allowing to obtain a derived verifiable credential from an eIDAS Identity Provider.
- Chapter 4: "T3.2 Universal Ledger Resolver" describes the various components that the KRAKEN project has developed to facilitate the alignment with the guidelines and services provided by ESSIF.
- Chapter 5: "T3.3 Edge Tools for the Decentralised Identity Solution" defines the components that will be used by the external actors of the SSI solution. The main developments are a mobile application (Ledger uSelf Mobile application) and a server REST API (Ledger uSelf Broker).
- Chapter 6: "T3.4 SSI Wallet Management" outlines the mechanism necessary for managing the SSI key material and the VCs through a Backup system, allowing to backup and synchronise the SSI information, restoring these data on another devices, in the case the end users lose their identity information.

2 SSI Solution overview

Under the umbrella of a Self-Sovereign Identity solution, the KRAKEN project has developed a series of components and subcomponents. Figure 1 below shows at a glance the main components produced in WP3 depicted in different colours to facilitate the mapping between components and the task where they were developed.

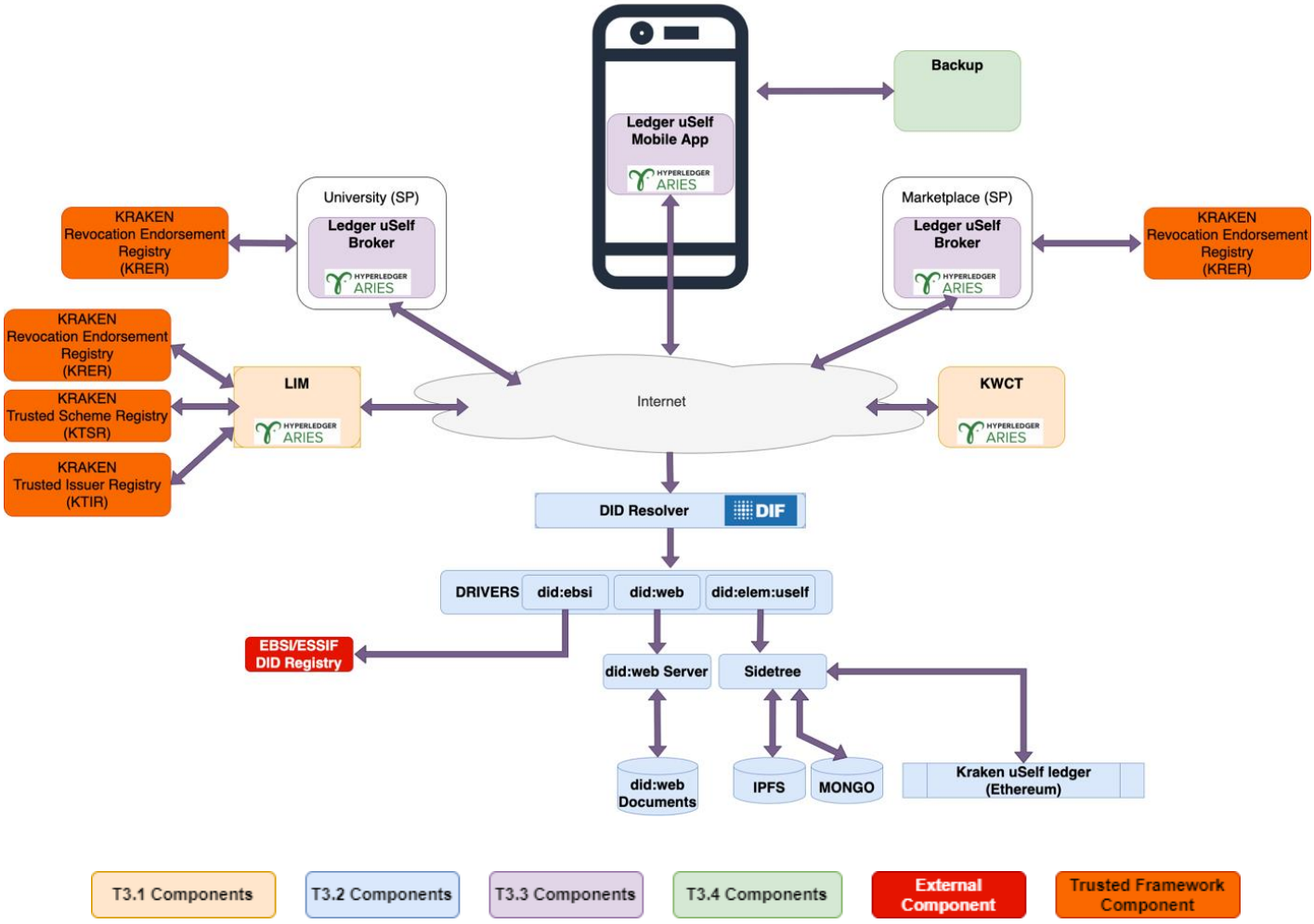

Figure 1: SSI components overview

As general consideration and as one of the main improvements from the previous release, this new set of components support the management of public DID, providing the associated services for creating/generating, publishing and then using the public DIDs for the different use cases and scenarios, including external Distributed Ledger Technologies (DLT) as EBSI/ESSIF information [3].

As can be seen in Figure 1, the components have been classified by colours. Components in yellow, LIM and KWCT are described in detail Chapter 3, components supporting public DIDs management (in blue) and alignment to EBSI/ESSIF (in orange) are defined in Chapter 4. The edge components, represented in purple are described in Chapter 5 and finally Chapter 6 describes the Backup system, components in green.

3 T3.1 Verifiable Credential management tools Components

KRAKEN implements two specific components for Verifiable Credential Management: The Legal Identity Manager and the Web Company Tool. Their purpose and functionalities are described in subsections 3.1 and 3.2 below.

3.1 Legal Identity Manager (LIM)

From the submission of the deliverable D3.1 [4] the LIM changed only in the underlying version of the Hyperledger Aries Framework [2] of the Rest Go Agent, that moved from V0.1.6 to V01.7, and in some usability enhancement connected to a better integration of the User Interface with the asynchronous events raised by the Aries Rest Agent.

A major part of the descriptions and the technical schemas contained in D3.1 are still valid. Some parts have been updated.

3.1.1 Description

The Legal Identity Manager (LIM) is a service offered on the web that provides to European Citizens (only natural persons, not legal persons) a verifiable ID, named e-ID (electronic ID) derived from an eIDAS identity assertion. Additionally, it allows European citizens to sign documents with a signature certificate derived from their eID.

The LIM acts in two different capacities:

- As an SSI issuer, for issuing an eID to a citizen. To this purpose, the LIM implements an “eIDAS service provider”, which connects to the Italian eIDAS eID node. Through the network of eIDAS EU nodes, an EU citizen can authenticate using her national eID mean.
- As an SSI verifier, for allowing a user to sign an arbitrary document by presenting her e-ID. In this scenario, a user submits a PDF document to be signed, together with her eID. The service validates the eID and then invokes the InfoCert remote signature service. This service generates a one-shot signing key pair and a one-shot/time (qualified) certificate and use them to sign the PDF document with PDF Advanced Electronic Signatures (PAdES), the most widely adopted type of digital signature.

The LIM comprises the following components:

1. LIM_WebSite: the web site of the tool
2. LIM_Backend: server-side component that acts as an eIDAS SP, it manages the eIDAS authentication phase
3. LIM_ExpressWebServer: a Web server that acts as an HTTP proxy and webhook listener
4. LIM Database: standard database
5. LIM GoRestAgent: standard Aries Go REST agent v0.1.7

Figure 2 below depicts the LIM components, the interactions with the Italian eIDAS node (eIDAS_AGID_Proxy) and InfoCert remote signature service.

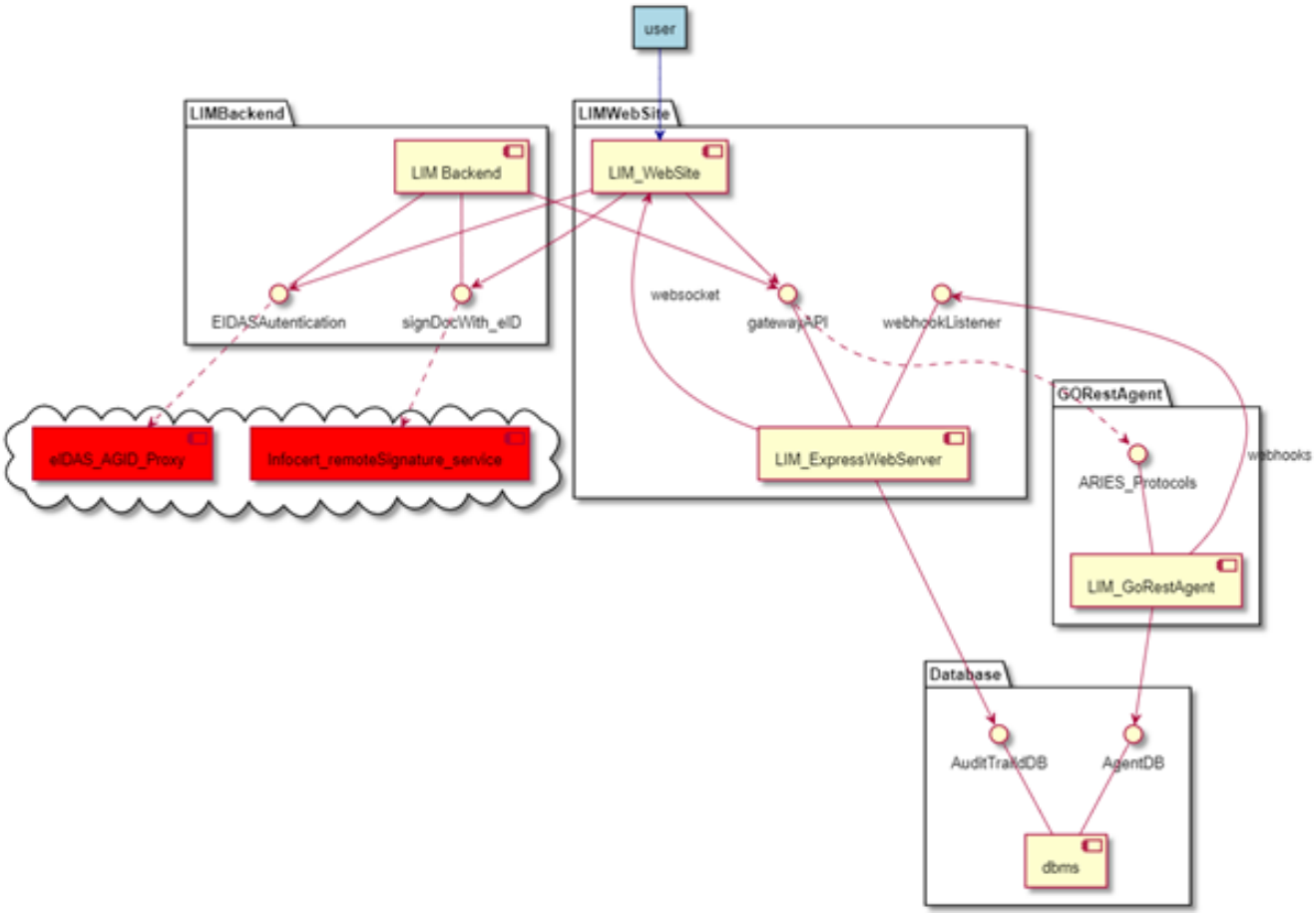

Figure 2: LIM Components and interactions

e_ID issuing sequence diagram

The eID issuing is a semi-automatic asynchronous process organized in two main phases:

1. The SSI DID exchange process that enables the LIM menu
2. The user's request of an eID in the UI of the LIM:
    1. The eIDAS authentication process
    2. The issuing of the eID VC by the LIM

The entire process is described in the sequence diagrams provided in Figure 3, Figure 4 and Figure 5.

In the diagrams we used solid lines to represent REST API calls, dotted lines to represent webhooks, HTTP redirects or actions of the user on the UI, to distinguish between what is synchronous from what is asynchronous.

Figure 3 provides the entire DID exchange main phase 1: It is a standard Aries DID Exchange protocol triggered by the user that visits the LIM landing page to use or setup a DID connection with the LIM agent.

The holder tool is the SSI tool that will contain the eID_VC after the issuing by the LIM, typically it is a mobile app. The holder agent is the Aries Go REST agent used by the holder tool. The Aries Go REST agent implementation includes an "AUTOACCEPT" feature, which streamlines the flow.

Figure 4 presents the Sub-Phase 2.1, which starts when the user chooses the eID functionality. The LIM acts as an eIDAS relying party, communicating with the national eIDAS node, which gives access to the European eIDAS network.

The involved eIDAS node component is the Italian agency for digitalization “AGID SP Proxy”.

The red components in the sequence below are external to the LIM, they have been included for completeness of description of the full authentication process, and they are:

- AGID SP Proxy: it acts as a proxy of the eIDAS service providers of the different countries. The user chooses on the UI which country he uses to authenticate.
- WebPageCountry: it represents the web page of the country’s eIDAS service provider the user is redirected on after the choice of the country.
- EIDAS_NODE_WEB: it represents the eIDAS node of the chosen country.

Finally, Figure 5 shows the sub-phase 2.2. After the reception of the user eIDAS assertion, the LIM produces, signs and delivers the eID. The VC issuing process follows the Aries protocol Issue Credential’s flow [11]. The LIM, as an issuer, sends a proposal to the holder, then an agent-to-agent communication between the LIM’s agent and the holder’s agent takes place.

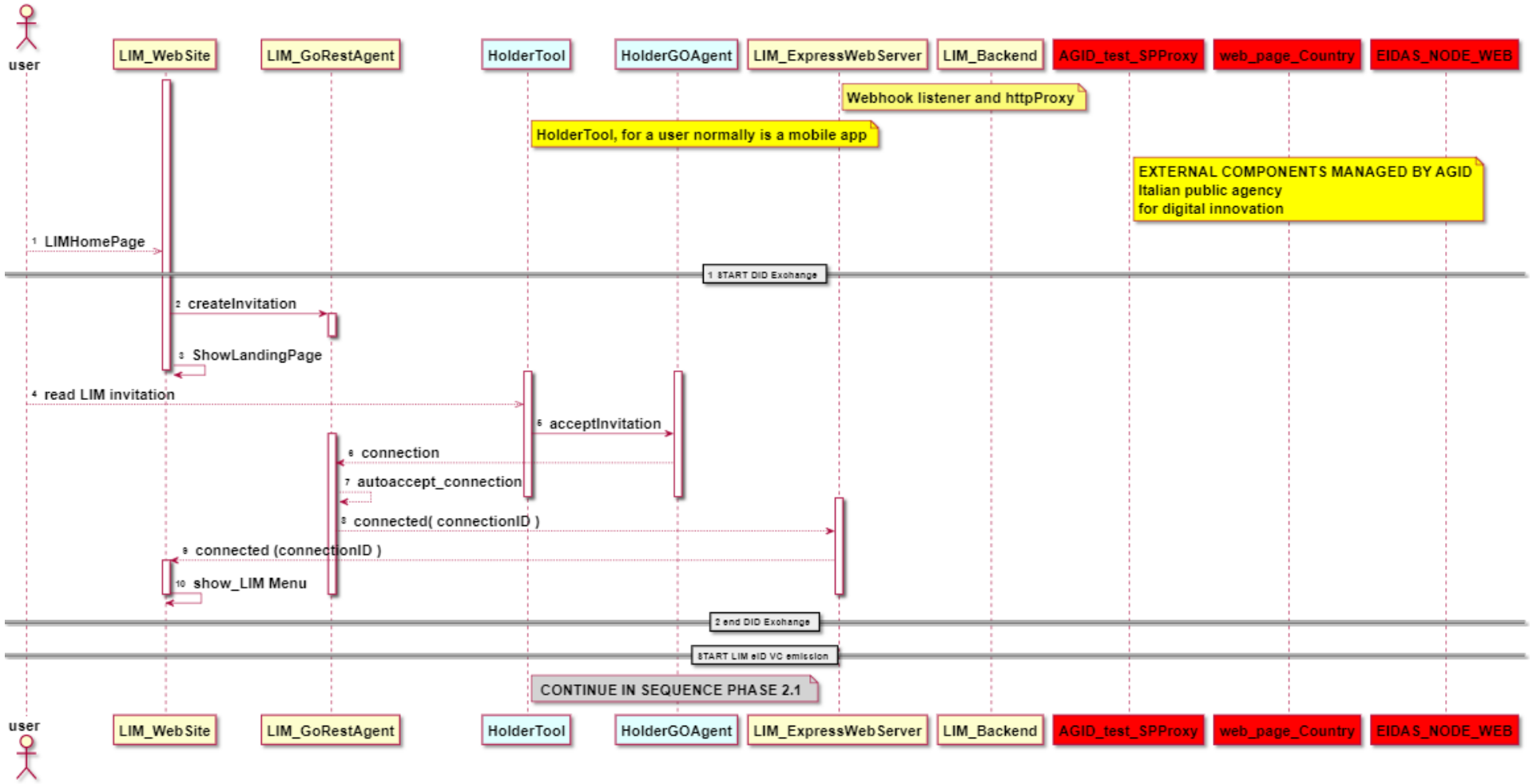

Figure 3: DID exchange phase

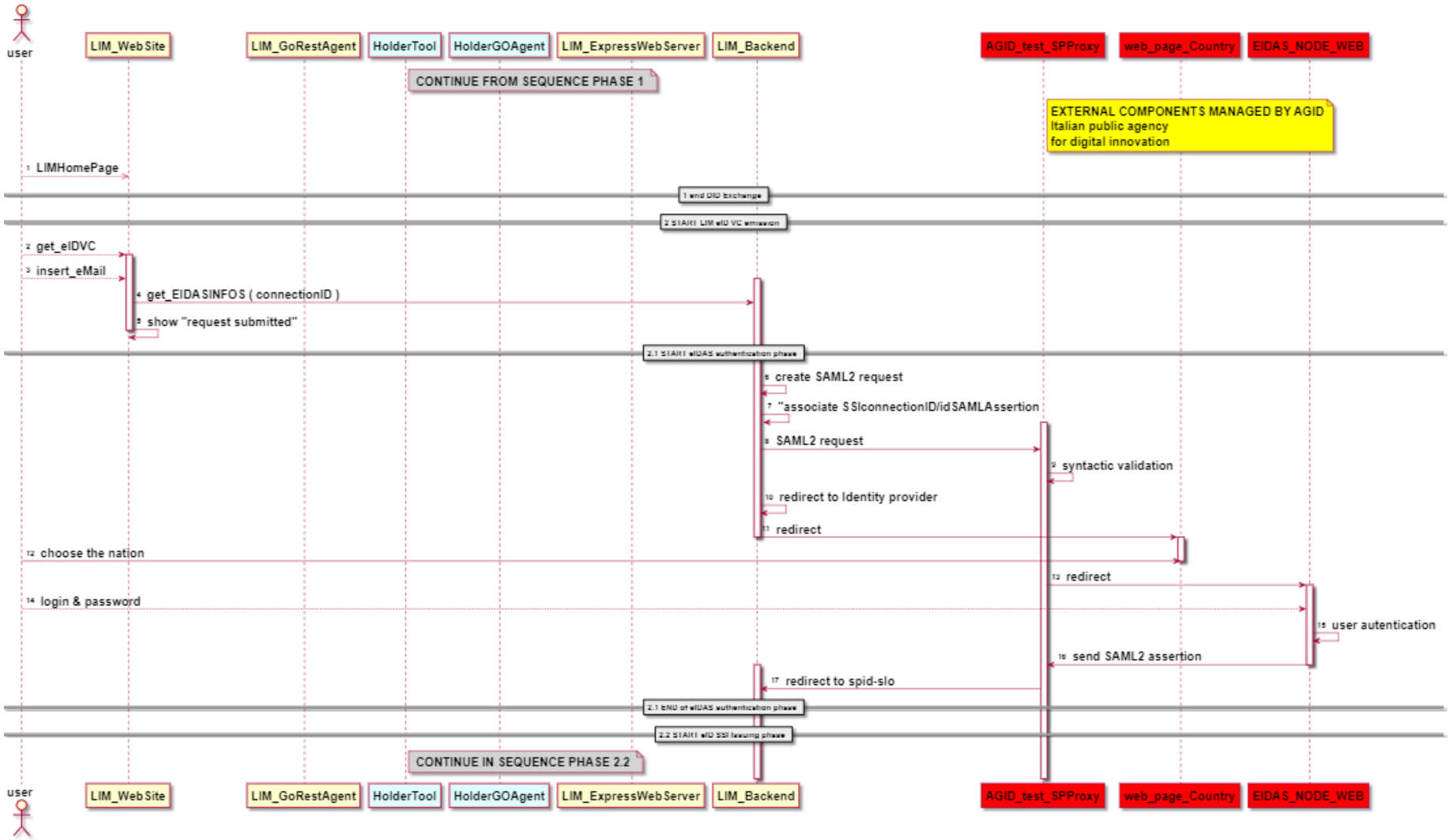

Figure 4: eIDAS authentication phase

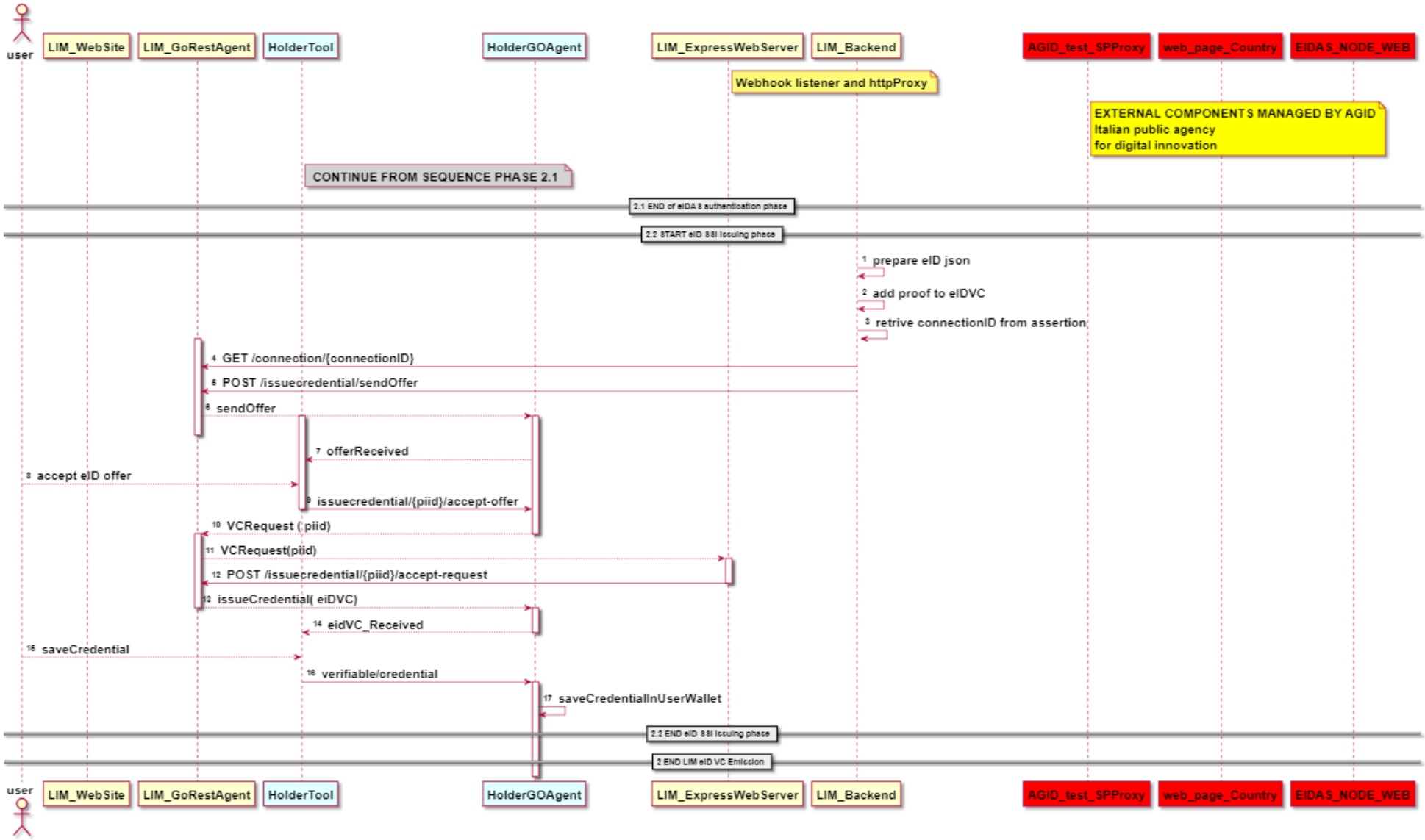

Figure 5: Issuing of the eID VC

3.1.2 Interfaces

The described components expose the following interfaces:

- LIM_WebSite: A web user interface for accessing LIM asset has been developed. Figure 6 shows a screenshot of the LIM entry page.

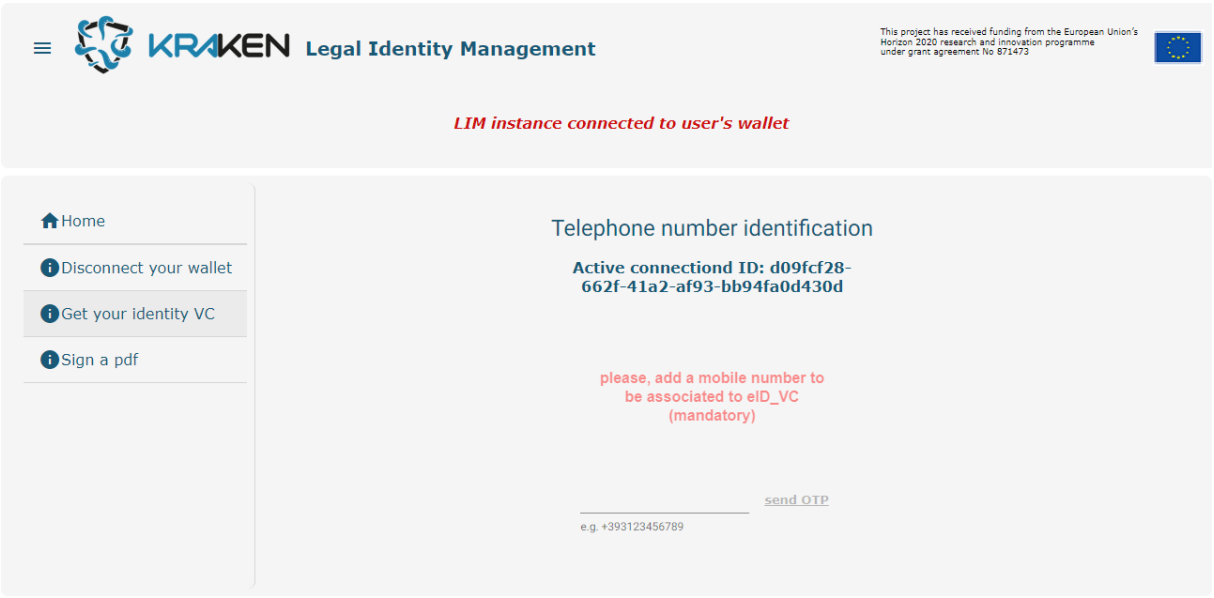

Figure 6: LIM web interface

- LIM_Backend:
  - eIDAS_auhentication: based on HTTP-redirect to the AGID's eIDAS Italian Node using SAML2 protocol
  - SignDocWith_eID: the API used to sign pdf documents with advanced signature.
    - token (POST): Get the OpenID Connect token:

      https://idpstage.infocert.digital/auth/realms/delivery/protocol/openid-connect/token
    - sign (POST): Sign the xml security assertion deriving from the eID credential in Infocert format

      https://apistage.infocert.digital/signature/v1/certificates/K8E0D1OQP1B1/sign
    - challenge (POST): enrol the one-shot certificate for the identity used to sign the document

      https://apistage.infocert.digital/signature/v1/authenticators/SMSP/challenge
    - oneshot/sign (POST): return the signed document

      https://apistage.infocert.digital/signature/v1/oneshot/sign
- LIM_ExpressWebServer:
  - gatewayAPI: acts as an http proxy on the following Aries protocols [9] [11] [12]:
    - DID Exchange
    - Issue-Credentials
    - Present Proof

- webhookListener: a private to LIM exposed standard web hook listener called in POST by the LIM's Aries Go agent.

- LIM_dbms: agentDB
  - Private to LIM exposed standard interface of CouchDB dbms.
- LIM_GoRestAgent: Private to LIM exposed standard Aries_Go agent protocols used in KRAKEN [9] [11] [12].

3.1.3 Deployment

The LIM is deployed in the following five docker containers, as represented in Figure 2.

1. LIM_WebSite
2. LIM_express_webserver
3. LIM_backend
4. LIM Database
5. LIM GoRestAgent

3.1.4 Source code

The source code of these components can be found in the following private KRAKEN GitHub repositories:

https://github.com/krakenh2020/LIM_SP_eidas_backend

https://github.com/krakenh2020/LIM_httpProxy

https://github.com/krakenh2020/LIM_AngularFrontEnd

3.1.5 Baseline technologies and tools

Excluding the Aries framework, which has been extensively describes, the only baseline technology adopted in LIM is the open-source tool "spid-go" [16] provided by the "Team per la Trasformazione Digitale - Presidenza del Consiglio dei Ministri" (the Digital Transformation Team - Presidency of the Council of Ministers of the Italian Government). The LIM backed acts as an eIDAS relying party based on "spid-go".

3.2 KRAKEN Web Company Tool (KWCT)

The KWCT was delivered upon presentation of deliverable *D3.1 Self-Sovereign Identity Solution First Release* [4] and it has been fully described in that deliverable. From the submission of that deliverable the tool changed only in the underlying version of the Hyperledger Framework of the Aries Rest Go Agent, that moved from V0.1.6 to V0.1.7, and in some usability enhancement adding some descriptions in the elements of the user interface. The descriptions and the technical schemas contained in the deliverable D3.1 are still valid and will be used in the rest of this section changing only few parts when incorrect.

3.2.1 Description

The KWCT, former KRAKEN Web Business Logic in D2.4 [10], is a company "SSI general purpose" tool, i.e., a tool which exchanges SSI transactions on a variety of uses cases.

The KWCT operates on an identity wallet associated to a company, not to a specific user. Via this tool company users can manually run VC issuing / VC verification processes, when automation is not

possible, or to deal with cases which need manual intervention because they fall outside the normal processes (which are, presumably, automated).

To guarantee that only authorized company users can operate on the tool, both the Angular 13 front-end and the restful API used by the frontend, are protected; moreover, the underlying rest agent's rest API is not exposed. The KWCT security is delegated to an instance of Keycloak [14], an open-source identity and access management deployed on AWS. The KWCT user authentication and authorization is based on users and roles configured in Keycloak, the protocol used is OpenID Connect [15]. The tool works using the Aries's Go REST agent implementation; it supports the following Aries protocols:

- DID Exchange
- Issue-Credentials
- Present Proof

Issuing and revocation of credentials are based on the underlying framework functionalities (VC Schemas and VC revocation/endorsement). Since revocation/endorsement functionalities are not available in the current Aries-Go implementation, KRAKEN will provide a custom implementation.

The tool manages three kinds of entities:

- DID connections
- VCs
- Verifiable presentations (i.e., aggregation of claims extracted from verifiable credentials, as per W3C Verifiable credential model [17]).

Figure 7 depicts a deployment diagram illustrating the KWCT nodes and their interactions with the agents, currently each node is deployed in one or more Kubernetes container, in the future the containers organization could be simplified.

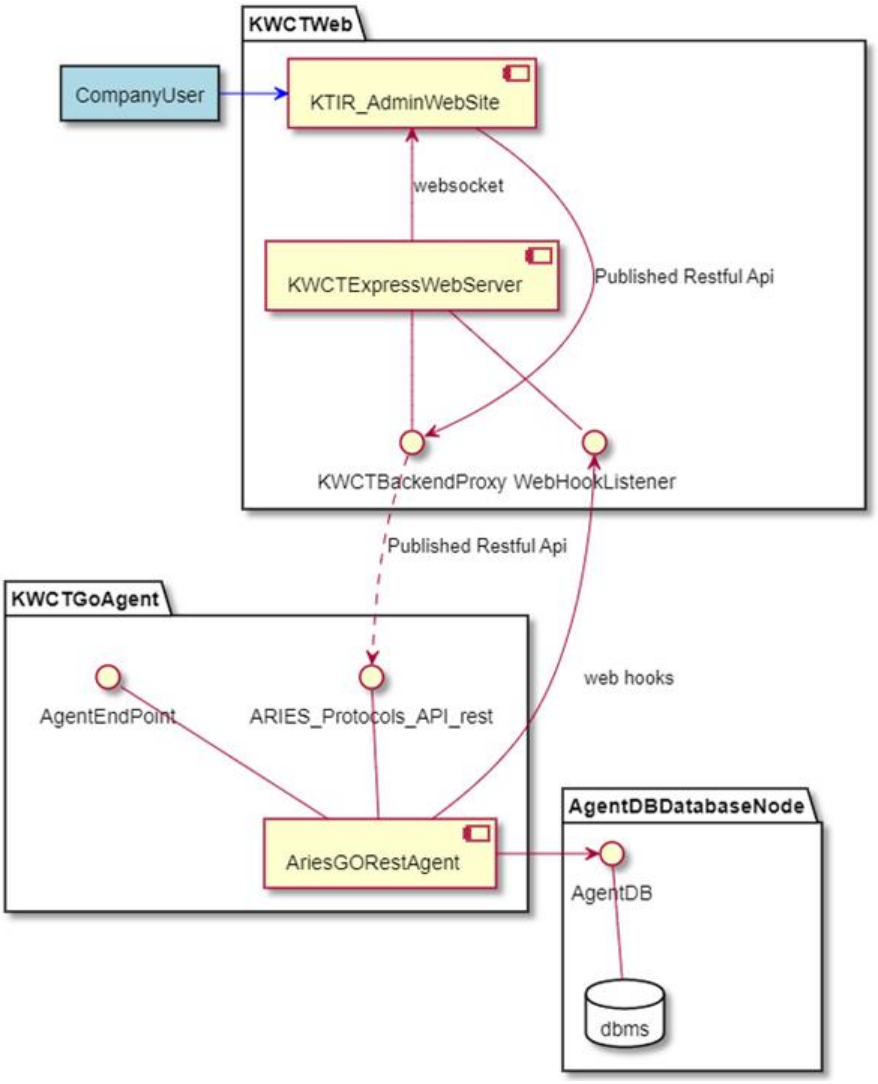

Figure 7: KWCT node and interactions

The software components in Figure 7 are:

- KWCT Web Node:
  - KWCTWebSiteAngular: standard angular 13 web site
  - KWCTexpressWebServer: an express framework nodejs web server acting as http proxy
- AgentNode:
  - Standard Aries Go Rest Agent v 0.1.7
- AgentDBDatabaseNode:
  - Apache CouchDB instance.

The KWCTWeb acts as a "controller" (i.e., the business logic) with regards to the KWCTGoAgent, according to the interaction flows which is part of Hyperledger Aries model [18]. The agent also leverages on a persistence layer granted by the AgentDBDatabaseNode.

3.2.1.1 Company User registration and authentication

There are two categories of users interacting with KWCT:

- External subjects (i.e., users who does not belong to the organization that deployed the KWCT). They will interact using the standard SSI protocols in order to connect, request and get verifiable credentials, issue credentials/presentations.

- Internal subjects (i.e., personnel belonging to the organization). They will be able to issue a credential or send a presentation using the KWCT Web Interface acting on behalf of the organization.

This last category of subjects will need to authenticate to the KWCT. Technically, the company's user authentication is managed using a standard Java Web Token-based (JWT) user session managed by Keycloak [14].

The internal user registration and authentication process is described in the sequence diagram shown in Figure 8. The user will use a web interface to perform a "signup" w.r.t a backend component (KWCTExpressWebServer). After that he will be able to perform a "signin". The server component will return a standard JWT token after successful authentication. The token will be issued for subsequent interactions.

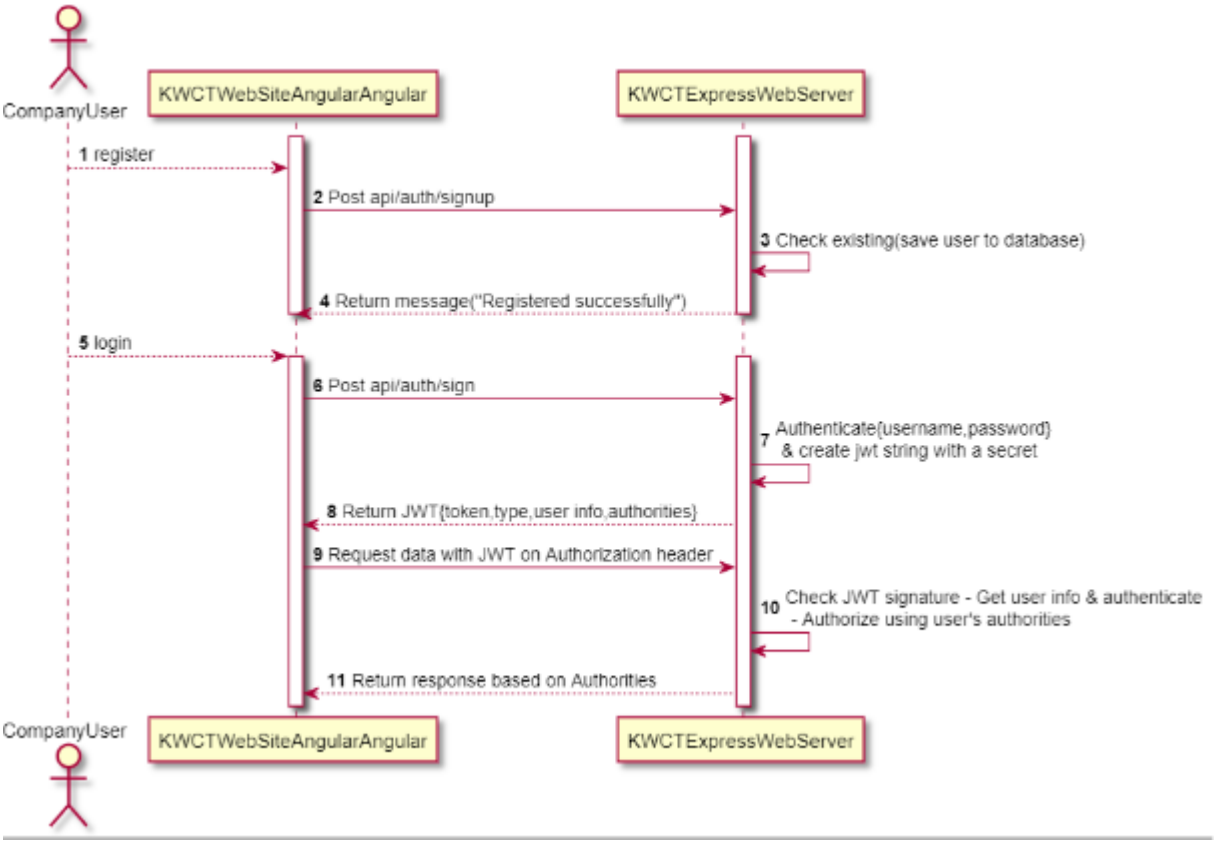

Figure 8: User registration and authentication

3.2.1.2 KWCT VC issuing

The VC issuing process implemented by the KWCT is a canonical issuing protocol, i.e., a process conformant to the Hyperledger Aries specification [11], driven by a company operator. The sequence is not represented since it would be largely a replication of **e_ID issuing sequence diagram** paragraph in Section 3.1.1.

3.2.2 Interfaces

The components expose the following interfaces:

1. KWCT_WebSite: A web user interface for accessing KWCT asset is developed. Figure 9 shows a screenshot of the KWCT tool and VC details.

2. KWCT_ExpressWebServer:
    - KWCBackendProxy: acts as a http proxy on the following Aries protocols [9] [11] [12]:
        - DID Exchange
        - Issue-Credentials
        - Present Proof
    - webhookListener: a private to KWCT exposed standard webhook listener (i.e., an http POST callback, according to a parttern which is widely used in the web development community) called by the LIM's Aries Go agent.
3. LIM_dbms:
    - agentDB: Private to LIM standard interface of CouchDB dbms.
4. KWCT_GoRestAgent: standard Aries_Go agent protocols used in KRAKEN [9] [11] [12].

The tool is provided as a web interface. The REST API exposed by the KWCT based on the Aries protocols DID Exchange, Issue credentials and Present Proof, is not public and only the LIM front-end uses it.

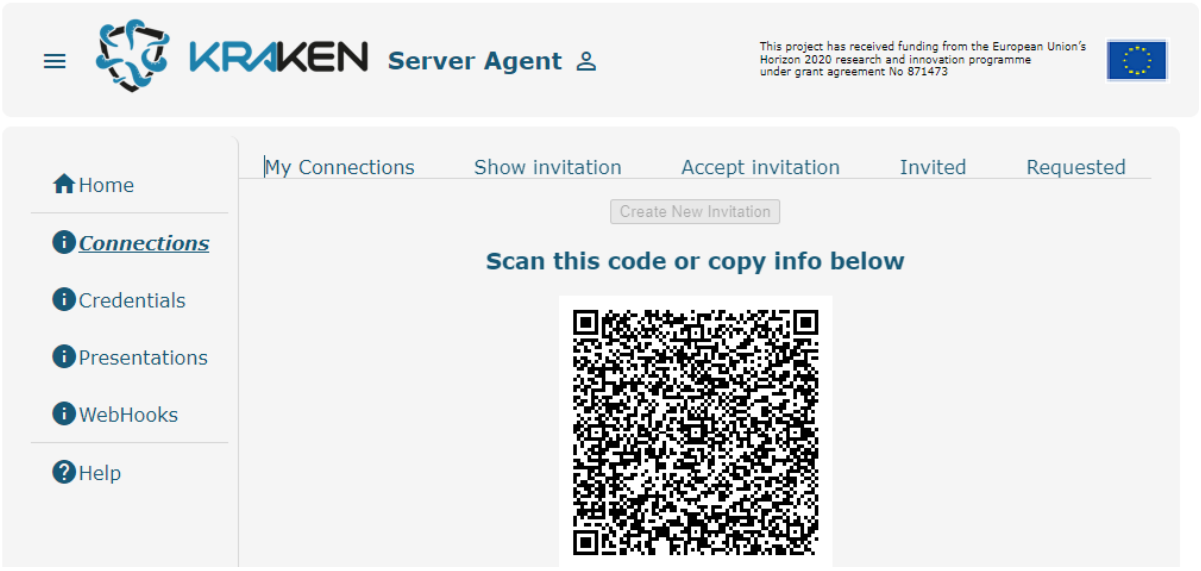

Figure 9: KWCT web interface and VC details

Figure 9 shows the web interface of the KWCT, namely the tool is open on the view "Connections", activated by the left side menu "Connection". This view supports the protocol "DID_Exchange". The tabs in the upper part of the view, allow the user to operate on the connection objects, in the different states, as received by the KWCT tool instance. Using these tabs, a user can see the connections already active, to produce an invitation object, both in JSON and code format, or to accept incoming connection invitations and so on.

3.2.3 Deployment

The KWC is deployed as three containers in Kubernetes on AWS, these containers are described in Figure 9:

1. KWCTWebSiteAngular
2. KWCTexpressWebServer
3. KWCT_GoRestAgent

The DBMS node is common with other tools deployed on AWS.

3.2.4 Source code

The source code of these components can be found in the following private KRAKEN GitHub repositories:

https://github.com/krakenh2020/KWCTAngularFrontEnd

https://github.com/krakenh2020/KWCT_httpProxy

3.2.5 Baseline technologies and tools

The KWCT derives from an open-source example in Angular 8 provided by Hyperledger community for Cloud Agent Python implementation (ACA-Py) [19], based on the Indy Ledger.

4 T3.2 Universal Ledger Resolver

As indicated in the previous iteration of this document this section describes the different services devoted to supporting trustability in the Self-Sovereign Identity solution. On one hand, this section describes the Trusted Framework which compiles several different trusted registries developed according to the guidelines provided by EBSI/ESSIF. On the other hand, the Public DID Infrastructure subchapter describes the different services devoted to supporting access to the DLTs used by the Self-Sovereign Identity solution implemented by the KRAKEN project.

4.1 Trust Framework

The final design of the Trusted Framework (Figure 1) comprises the KRAKEN Trusted Issuer Registry (KTIR), the KRAKEN Trusted Schema Registry (KTSR), the KRAKEN Revocation Endorsement Registry (KRER) and the KRAKEN DID Registry (EBSI/ESSIF DID Registry in Figure 1). These components are functional and ready to be integrated into the KRAKEN platform. Also, it is worth mentioning that the services described in this section follow the EBSI/ESSIF guidelines and therefore it will make a future adoption/integration of the EBSI/ESSIF services easier once they will be publicly available.

4.1.1 KRAKEN Trusted Issuer Registry (KTIR)

In an SSI environment, cryptographic proofs allow verifying that Verifiable Credentials are formally correct and have not been tampered with, but they cannot grant that a credential's issuer is trustworthy. In order to trust a verifiable credential, a relying party (verifier) needs to accept the issuer of the credential as a trust anchor. To make this model scalable, KRAKEN follows EBSI approach by defining a registry of trusted issuers (KRAKEN TIR, or KTIR). The registry includes the list of issuers which are supposed to be trusted in KRAKEN environment. The registry is maintained by a specific entity assumed to be trusted, and open to any verifier.

To guarantee that only authorized maintainers can operate on the registry, both the admin web site and the restful API used by the site, are protected.

As for KWCT, KTIR access control is delegated to an instance of Keycloak [14] , an open-source identity and access management deployed on AWS. The KTIR user authentication and authorization is based on users and roles configured in Keycloak, the protocol used is OpenID Connect [15]

Efforts have been made to reflect part of the public APIs of ESSIF v2 TIR component.

4.1.1.1 Description

KRAKEN TIR's architecture is organized in three tiers:

- KTIR_AdminWebSite**:** An angular 13 web tool dedicated to KRAKEN's platform administrator users, used to populate the KTIR's_database. The access to the tool is protected by Keycloak using OpenID Connect. The tool invokes the KTIR_backend services through KTIR_private interface.
- KTIR_backend**:** A server-side component that exposes two REST APIs:
  - KTIR_Private_CRUD_api: protected by Keycloak and accessible only by the KTIR_AdminWebSite, it provides Create, Read, Update and Delete (CRUD) functionalities on the KTIR_database entities.
  - KTIR_public_Api: published to internet, provides methods to be used by SSI verifiers.
- KTIR database: Is the container of the configuration info of the KTIR.

Figure 10 below shows the KTIR components described in this section.

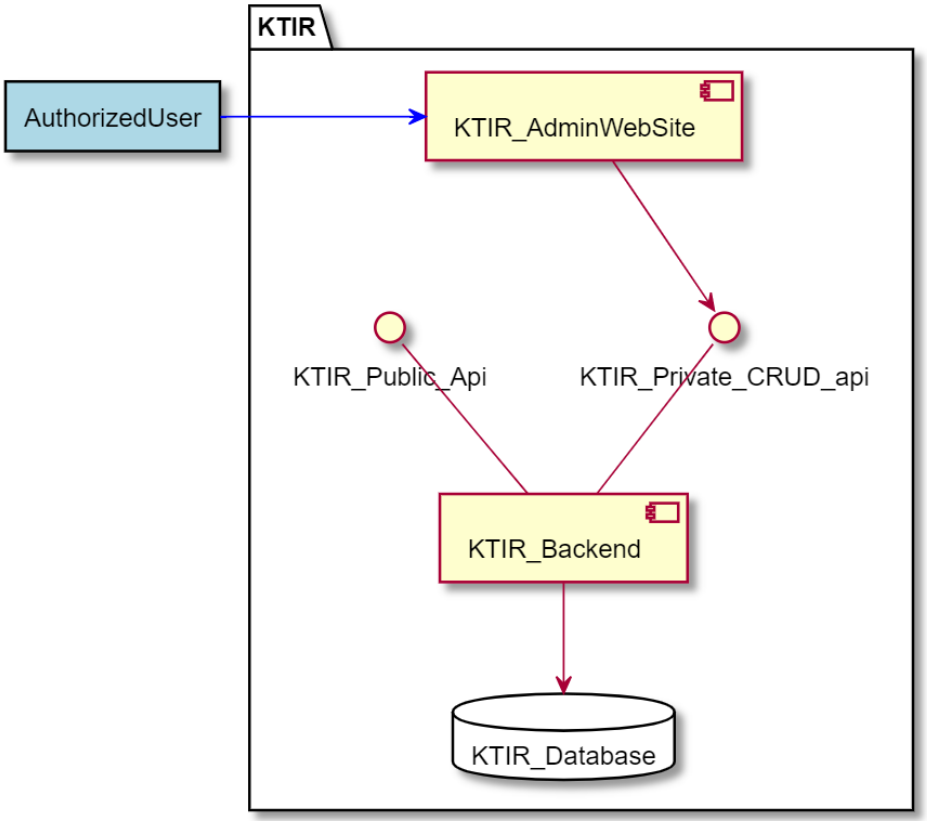

Figure 10: KTIR components

4.1.1.2 Interfaces

The described components expose the following interfaces:

- KTIR_AdminWebSite: A private technical web user interface added to manage the TIR's data. Figure 11 below shows a screenshot of the private KTIR's web interface. The interface, after user authentication, offers the functionalities to operate as an SSI issuer and verifier.

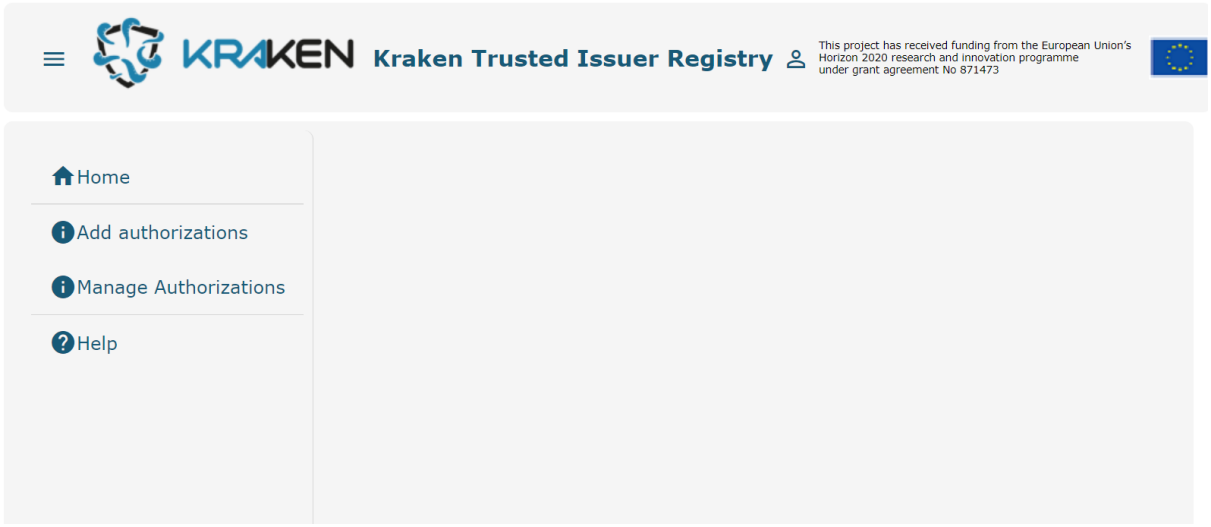

Figure 11: KTIR web interface

- KTIR_Public_API: This API implements a single method that permits to verify that an issuer is authorized, in KRAKEN, to issue credentials belonging to a schema. The API is described in the CURL command in Figure 12 below:

```
curl --location --request POST 'https://serverURL/ktir/isIssuerAuthorized'
--header 'Content-Type: application/json'
--data-raw '{"did" : "a DID","schema_uri" : "a schema URI"}}
```

Figure 12: KTIR public API

4.1.1.3 Deployment

The KTIR tool will be deployed as two docker containers as represented in Figure 10.

4.1.1.4 Source code

The source code of these components can be found in the following private KRAKEN GitHub repositories:

https://github.com/krakenh2020/KTIR_angularFrontEnd

https://github.com/krakenh2020/KTIR_httpProxy

4.1.2 KRAKEN Trusted Schema Registry (KTSR)

The Verifiable Credential's schema defines the content (set of claims) of the credential that belongs to that schema. The SSI Hyperledger Aries GO implementation [2] chosen in KRAKEN is not providing this functionality, so KRAKEN consortium decided to implement a minimal solution to support the schema definition/verification. On this base, schemas will be represented in JSON schema draft-07 format [13] and published in a Git repository. Each user can validate a JSON according to the schema (a jsonschema.org standard format[1]). These schema files are used by the KWCT as specifications of the VC claims structure to generate the forms to add values to the VC claims.

4.1.2.1 Description

The KTSR is published in a Git repository with read access, that contains:

- An "index" JSON file that contains the list of the VC schemas used by the system
- One JSON file (aligned to the JSON schema draft-07 specification [13]) for every VC's schema used by a KRAKEN issuer to issue a credential used inside KRAKEN.

Figure 13 below presents the structure of the index JSON file containing the list of schemas.

[1] https://json-schema.org/

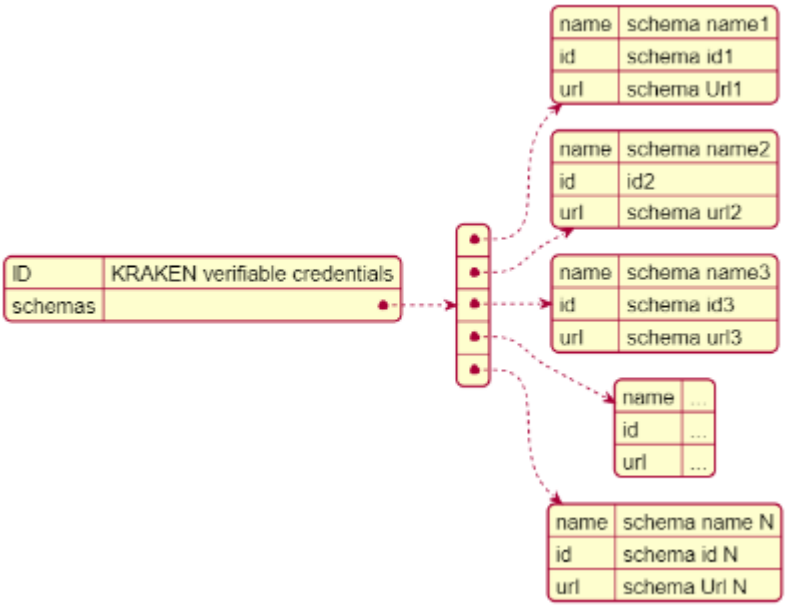

Figure 13: JSON containing the list of the schemas managed by KRAKEN

4.1.2.2 Deployment

The schemas are hosted in GitHub at the following url:

https://github.com/krakenh2020/vc-schemas

4.1.2.3 Schema examples

The figures in this sub-section (Figure 14 and Figure 15) present a simple example of a working[2] JSON VC schema document that specifies the organization of the credentialSubject of a sample DegreeVC using the draft-07 specification from the JSON-Schema.org [13].

In this example, the only requirements to be satisfied are that a degreeCredential must contains 3 claims: “id”, “degreeType” and “degreeName” and that, each of them, is of type string.

The draft-7 standard provides the means to specify stricter controls over schema components using regular expression or set of values etc. It is possible to activate controls by declaratively adding them to the VC schema definition file.

[2] Verified with the online tool: https://jsonschemalint.com/#!/version/draft-07/markup/json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "description": "Kraken sample degree verifiable credential sample",
  "credentialSubject": {
        "type": "object",
        "properties": {
            "id": {
                "type" : "string"
                  },
            "degreeType": {
                "type" : "string"
                  },
            "degreeName": {
                "type" : "string"
                  }
                }
    }
}
```

Figure 14: Example of VC schema json working document

```
{
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "id": "http://example.edu/credentials/3732",
    "type": ["VerifiableCredential", "UniversityDegreeCredential"],
    "issuer": "https://example.edu/issuers/14",
    "issuanceDate": "2010-01-01T19:23:24Z",
    "credentialSubject": {
      "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
      "degreeType": "BachelorDegree",
      "degreeName": "Bachelor of Science and Arts"
    },
    "credentialSchema": {
      "id": "https://the-url-of-the-json-VCScema-file.json",
      "type": "JsonSchemaValidator2018"
    },
    "proof": {
        "created": "2019-09-27T06:26:11Z",
        "creator": "did:work:MDP8AsFhHzhwUvGNuYkX7T#key-1",
        "nonce": "0efba23d-2987-4441-998e-23a9d9af79f0",
        "signatureValue": "2A7ZF9f9TWMdtgn57Y6dP6RQGs52xg2QdjUESZUuf4J9BUnwwWFNL8vFshQAEQF6ZFBXjYLYNU4hzXNKc3R6y6re",
        "type": "Ed25519VerificationKey2018"
    }
}
```

Figure 15: Example of DegreeVC used to test the VC schema document

4.1.3 KRAKEN Revocation & Endorsement Registry (KRER)

The Revocation Endorsement Registry component (KRER) has been developed following the basic information provided by the ESSIF reference architecture documentation [5]. At this moment this service is not provided as an EBSI building block [6] yet, thus the Consortium decided to develop a service that mimic its use.

4.1.3.1 Description

The KRER component contains information about the verifiable credentials (or Verifiable Presentations) necessary for their verification. Among other this registry stores attributes such as status (valid, revoked, suspend), and issuance or revocation dates, as described in *D2.5 KRAKEN final*

technical design [7]. The access to the services for creating or updating (POST and PUT) the credential status will be secured by an allowed list of IPs or an IP range[3].

Figure 17 shows the KRER components:

- The KRER core, containing the offered REST API services and services that can configure to control the access to the services based on a IP request filter.
- The Credential Status Storage for storing the credential status.

4.1.3.2 Interfaces

Based on the specifications described in *D2.5 KRAKEN final technical design* the following interfaces (Figure 16) have been implemented:

- credential/add (POST): Method for creating the status of a new credential.
- credential/update (PUT): Method for updating the status of an existing credential.
- credential/status (GET): An open and public method for finding the credential status providing the credential identifier.

As mentioned above, the credential/add (POST) and credential/update (PUT) interfaces have a security filter to control the access based on an IP filter mechanism. Only allowed IP addresses can access the revocation services to add or update the credential status. On the other hand, the access to the credential/status (GET) is always public and anybody can access to obtain the status of a verifiable credential. The following figure shows the details of the interfaces defined:

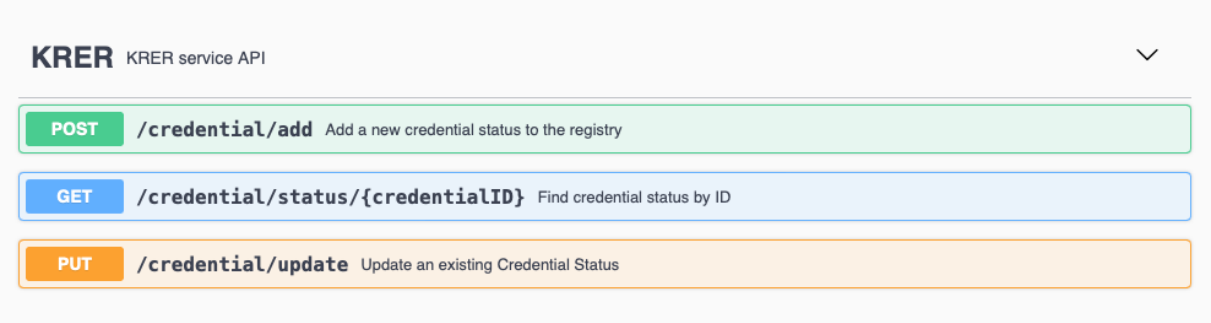

Figure 16: KRER Interfaces

4.1.3.3 Deployment

The deployment of the KRER component is envisaged to be done by each VC issuer (the SP) within their own facilities, using a docker image. To this end, it is necessary to configure the exposed entry port and the IPs (or IP range) where the server will accept requests. Figure 17 depicts the KRER deployment on the Service provider (SP) side.

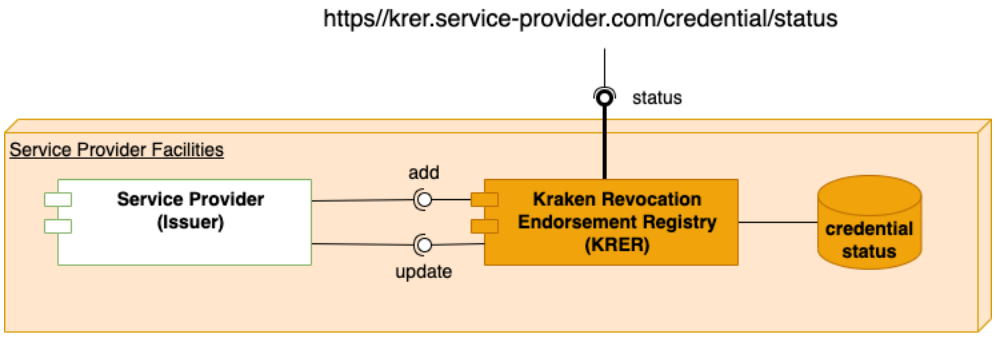

Figure 17: KRER deployment diagram

[3] https://www.baeldung.com/spring-security-whitelist-ip-range

4.1.3.4 Source code

The source code of this component can be found in the following private KRAKEN GitLab repository:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-ledgeruself-krer

4.1.4 KRAKEN DID Registry

The KRAKEN DID Registry is a service that manages Decentralised Identifiers (DIDs) stored into the EBSI blockchain infrastructure. It resolves EBSI compliant DIDs[4], relying on the W3C DIDs standards[5].

4.1.4.1 Description

This component was designed to provide the same functionality that the interface defined by EBSI/ESSIF which was not publicly available by then. The KRAKEN project wasn't selected to participate in the Early Adopter Programme promoted by EBSI/ESSIF.

When the service was released for public access the Consortium considered not necessary to implement it anymore but directly used the provided services by ESSIF. See the following link: https://app.preprod.ebsi.eu/console/

In Figure 1, the reader can see how the Self-Sovereign Identity Solution interacts with the EBSI/ESSIF DID Registry (in red).

4.1.4.2 Interfaces

This service has only one interface (Figure 18) for resolving the identifier by its DID:

https://api.preprod.ebsi.eu/docs/#/DID%20Documents/get-did-registry-v2-identifier

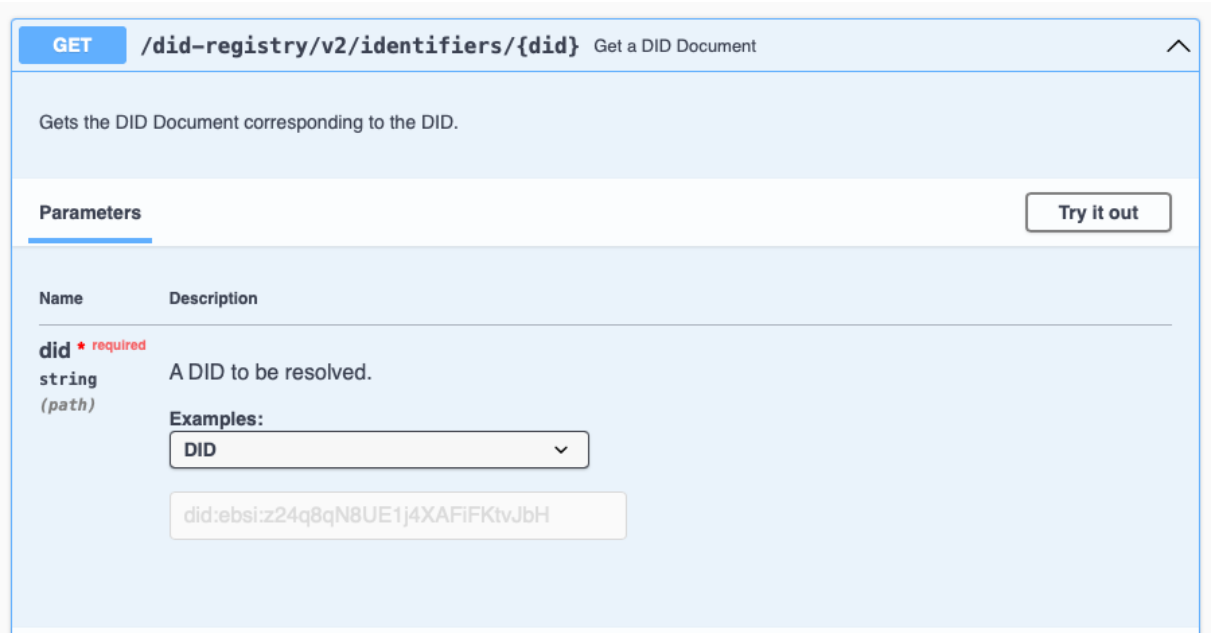

Figure 18: DID Registry Interface

The main aim of this method is to provide access to the DID document associated to a DID stored by ESSIF [4].

[4] https://ec.europa.eu/digital-building-blocks/wikis/display/EBSIDOC/DID+Registry+API
[5] https://w3c.github.io/did-core/

4.1.4.3 Deployment

Not applicable to this component.

4.1.4.4 Source code

The source code of this service is not available.

4.2 Public DID Infrastructure

The management of public DID and anchoring the associated DID documents into Distributed Ledger Technologies (DLT) is a fundamental part of the development of a Self-Sovereign Identity solution. This chapter describes the effort carried out in the KRAKEN project to provide the services that support this functionality. In this regard, it is worth mentioning that in a regular SSI operation there are two main use cases using these services: accessing/resolving the DID documents (for obtaining the cryptographic material on the normal course of the SSI flows) and creating/publishing these DID documents (including the creation and management of the cryptographic material associated with those DID documents). The Consortium has designed and implemented different services to resolve DID (top part of Figure 21) and create/publish DID (bottom components in Figure 21)

The latest updates of the W3C DID standard[6] introduce a new type of specification: DID methods. This new DID method concept defines the interaction with DLT regardless of the final technology used to store the information. As described in *D2.5 KRAKEN final technical design* [7], there is a new plethora of DID methods for different types of DLT (including the use of blockchain or using other types of decentralized systems). From those DID methods, the KRAKEN project selected three options that can be used by the Self-Sovereign Identity solution which are: did:ebsi, did:web and did:elem.

4.2.1 DID Registry

This section describes the main service for accessing/resolving the DID document associated to a public DID. The DID documents resolved by this component contain (among other information) the public keys that can be used in the normal flows of an SSI solution, such as validating a Verifiable Credential or helping to the establishment of a new connection.

4.2.1.1 Description

This service provides a single-entry point to DID document information regardless of the DID method or final DLT used to store the DID document. For that purpose an instance of the Universal Resolver has been deployed to be used by all the different components of the project. The Universal Resolver[7] is a project implemented by the Identity Foundation that resolves Decentralized Identifiers (DIDs) across many different DID methods, based on the W3C DID Core 1.0 and DID Resolution specifications[8]. The Consortium has also developed three different drivers, one for each of the DID methods supported by the KRAKEN project which are integrated in the final instance of the Universal Resolver, as the reader can see in Figure 21.

The outcome of this service is a DID Document. Figure 19 below shows a regular DID document served by this component.

[6] https://www.w3.org/TR/did-spec-registries/
[7] https://github.com/decentralized-identity/universal-resolver
[8] https://w3c-ccg.github.io/did-resolution/

```
{
  "@context":"https://www.w3.org/ns/did/v1",
  "id":"did:example:00001111",
  "authentication":[
    {
      "id":"did:example:00001111#z6MkecaLyHuYWkayBDLw5ihndj3T1m6zKTGqau3A51G7RBf3",
      "type":"Ed25519VerificationKey2018",
      "controller":"did:example:00001111",
      "publicKeyBase58":"AKJP3f7BD6W4iWEQ9jwndVTCBq8ua2Utt8EEjJ6Vxsf"
    },
    {
      "id":"did:example:00001111#_Qq0UL2Fq651Q0Fjd6TvnYE-faHiOpRlPVQcY_-tA4A",
      "type":"JsonWebKey2020",
      "controller":"did:example:00001111",
      "publicKeyJwk":{
        "kty":"OKP",
        "crv":"X25519",
        "x":"pE_mG098rdQjY3MKK2D5SUQ6ZOEW3a6Z6T7Z4SgnzCE"
      }
    }
  ],
  "service":[
    {
      "type":"NameOfService",
      "serviceEndpoint":"https://example.com/servicename/"
    },
    {
      "type":"NameOfService",
      "serviceEndpoint":"https://example.com/servicename/"
    }
  ]
}
```


Figure 19: DID Document example[9]

4.2.1.2 Interfaces

Figure 20 below shows the four interfaces provided by the public DID Registry available for the use of the KRAKEN project.

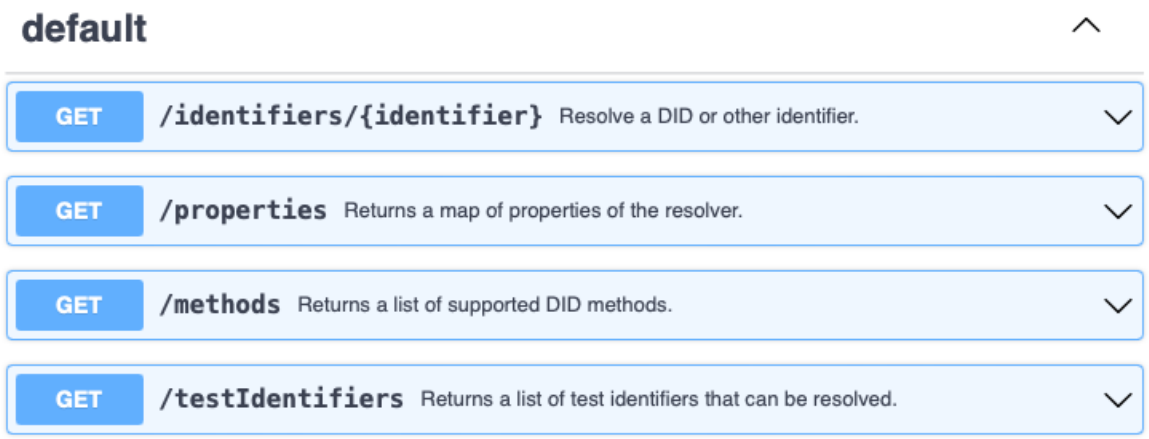

Figure 20: Universal DID resolver interfaces

9 https://api.preprod.ebsi.eu/docs/?urls.primaryName=DID%20Registry%20API#/DID%20Registry/get-did-registry-v2-identifier

These interfaces provide the following functionality:

- /identifiers/{identifier} (GET): returns a JSON file containing the DID document associated to the identifier (input parameter). An example of this outcome can be seen in Figure 19.
- /properties (GET): shows the configuration used for this Universal Resolver instance
- /methods (GET): returns the list of all the supported DID methods supported by this Universal Resolver instance
- /testIdentifiers (GET): for testing purposes this server contains some preloaded DIDs that can be used for testing the integration with other components.

4.2.1.3 Deployment

Figure 21 shows the whole view of solution implemented to provide a public DID infrastructure. It comprises the following components:

- The public DID resolver based on DIF [1] Universal Resolver[10] implementation, which redirects to the selected method used for resolving a DID;
- The DID Resolver Driver contains the different implementations to resolve the different DID methods[11] provided by KRAKEN, such as:
    - did:web for accessing a public domain;
    - did:uself for accessing the Sidetree infrastructure provided by KRAKEN;
    - did:ebsi for accessing the ESSIF/EBSI DID Resolver.
- The DID Web Server is a public server where the DID document is stored (see Section 4.2.2)
- The Sidetree Server (see section 4.2.3).

Regarding the deployment of the services to access/resolve the DID:

- DID resolver: serves the main entry point to access/resolve DID. It is publicly available using the following URL:

    https://resolver.prod.ari-bip.eu

- DID Drivers: contains the drivers used by the KRAKEN project. Each driver implemented handles the particular characteristics of the different DLT, and provides a common interface access regardless of the DID method used.

[10] https://github.com/decentralized-identity/universal-resolver
[11] https://www.w3.org/TR/did-spec-registries/

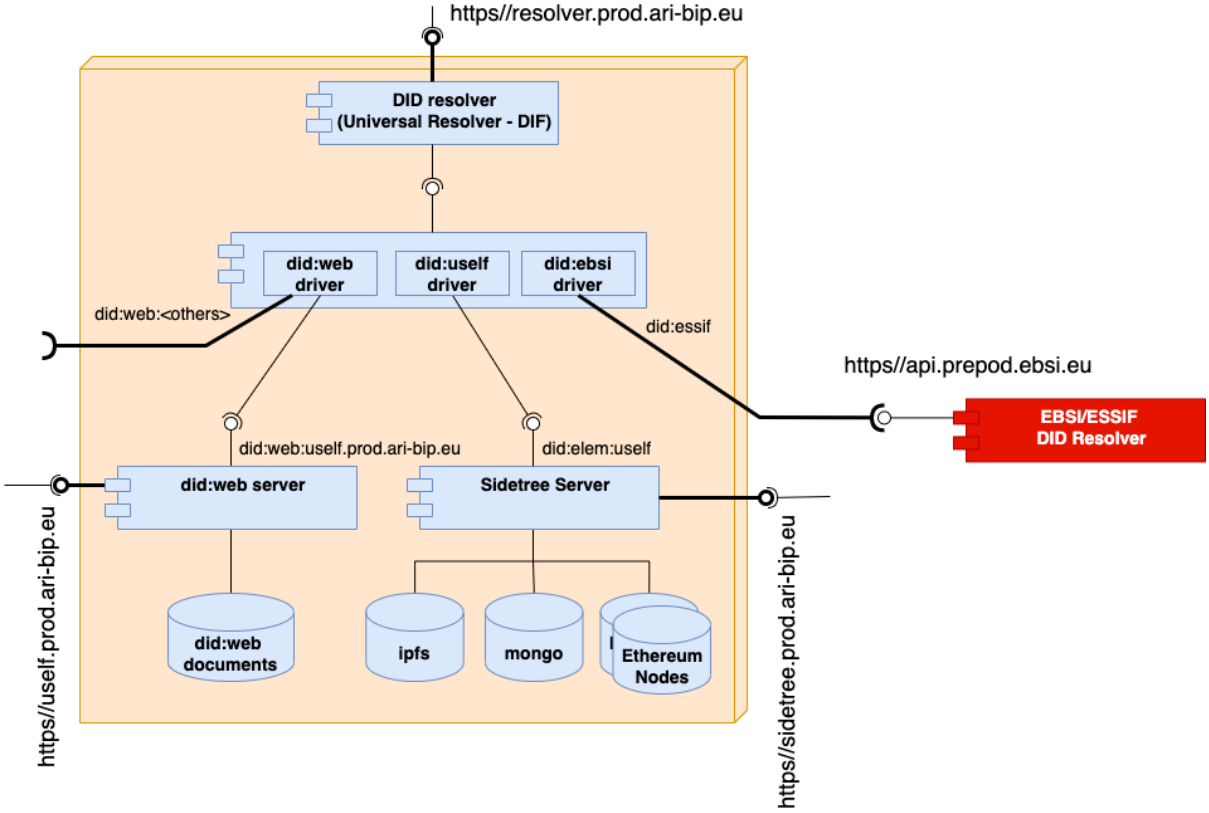

Figure 21: DID Universal resolver deployment

4.2.1.4 Source code

The DID Registry is composed of two components as the top part of the figure above shows, so the source code of the Universal Resolver component can be found in the following public GitHub repository:

https://github.com/decentralized-identity/universal-resolver

And the source code of the Universal Resolver Driver component can be found in the following private KRAKEN GitLab repository:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-ledgeruself-did-web

4.2.2 DID WEB server

As mentioned above there are different DID methods to manage the public DIDs and the information associated to them: methods to resolve the public DID and to create/publish them. This section of the document describes the implementation associated with the adoption of the did:web method in the KRAKEN project.

4.2.2.1 Description

This component enables to the KRAKEN project partners the use of a public server to publish DIDs following the did:web method. The did:web method is defined in *D2.5 KRAKEN final technical design* [7]. The adoption of this DID method implies that a blockchain infrastructure is not necessary, it applies the widely known Domain Naming System (DNS) standard to provide a decentralised storage solution.

4.2.2.2 Interfaces

Figure 22 below presents the three interfaces of the DID Web server.

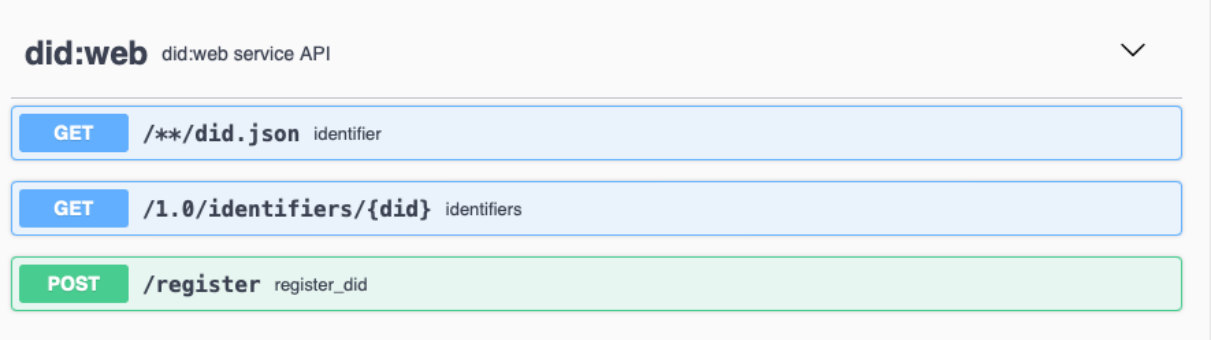

Figure 22: DID WEB server interfaces

These interfaces are:

- /**/did.json (GET): following the did:web method specification[12] this service provides the DID needed to request the associated DID document. See example of use: https://example.com/user/alice/did.json
- /1.0/identifiers/{did} (GET): this interface provides the DID document associated to the identifier (did) passed as parameter.
- /register (POST): this request allows storing (create and update) a DID document in the public server.

4.2.2.3 Deployment

Bottom left part of Figure 21 shows the deployment of the DID Web server and the relationships with the other components of the DID public infrastructure. This service is composed by two different components:

- DID Web Server: which provides a REST API interface. The detail of this interface is described in the previous section. This service is publicly available, and it can be accessed using the following URL:

  https://uself.prod.ari-bip.eu

- did:web documents: database devoted to storing the associated DID documents.

4.2.2.4 Source code

The source code of the DID Web Server component can be found in the following private KRAKEN GitLab repository:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-ledgeruself-did-web

4.2.3 Sidetree Server

This section details the second method implemented in the KRAKEN project to create/publish public DIDs, in this case following the did:elem's DID method. Further details about this DID method can be also found in the deliverable *D2.5 KRAKEN final technical design* [7].

[12] https://w3c-ccg.github.io/did-method-web/

4.2.3.1 Description

The Sidetree and DLT component provides access to the decentralized system, comprising Ethereum nodes, database (e.g. Mongo DB) or a IPFS system, as shown in Figure 21. The main difference with the did:web method described above, is that a blockchain infrastructure is used to store the information associated to the published DID documents. In addition, it is worth mentioning that the main interface to access this set of components is through an implementation of the Sidetree protocol[13]. The following sub-sections describe in detail the exposed interfaces and all the deployment details associated with this service functionality.

4.2.3.2 Interfaces

Figure 23 below presents the three interfaces of the Sidetree server.

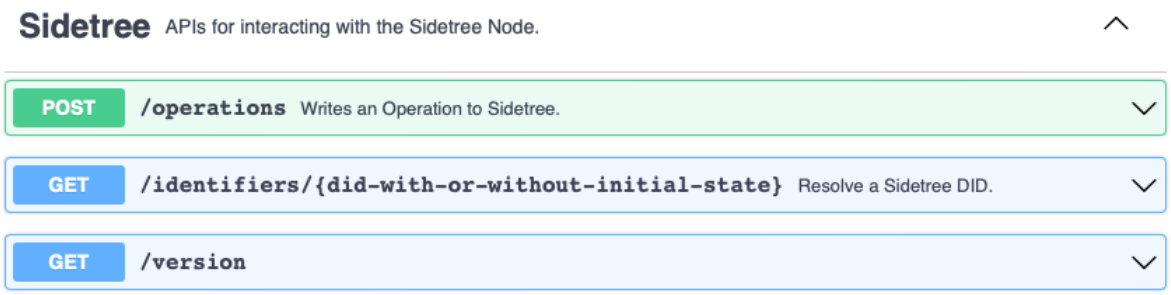

Figure 23: Sidetree and DLT interfaces

As mentioned before this service implements the Sidetree protocol[14] interface. The requests available are:

- /operations (POST): this request allows creating/updating/recovering and deactivating the DID documents within the ledger
- /identifiers/{did-identifier} (GET): provides the content of the DID document associated to the identifier passed as parameter
- /version (GET): this request returns the protocol version of the implementation used by the server

4.2.3.3 Deployment

The bottom right part of Figure 21 shows all the different components under the umbrella of the Sidetree server.

- Sidetree server: is the main interface of the system built following the Sidetree protocol. It is publicly available using the following link:

  https://sidetree.prod.ari-bip.eu

- MongoDB: this server stores the DID document
- IPFS server: the InterPlanetary File System (IPFS) is a server to store data in a decentralised approach.
- Ethereum nodes: provide a blockchain infrastructure to store the smart contracts.

[13] https://identity.foundation/sidetree/spec/
[14] https://identity.foundation/sidetree/spec/

4.2.3.4 Source code

The source code of the Sidetree Server component can be found in the following private KRAKEN GitLab repository:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-ledgeruself-sidetree

5 T3.3 Edge Tools for the Decentralised Identity Solution

The edge tools developed in the context of T3.3 comprise the Ledger uSelf Mobile app, the Ledger uSelf SDK, for a more friendly use of the Hyperledger Aries framework, and the ledger uSelf Broker.

5.1 Ledger uSelf Mobile app

The Ledger uSelf Mobile App is one of the main edge components, designed and implemented to allow the end users to use their own mobile devices to identify themselves following a Self-Sovereign Identity approach.

5.1.1 Description

This application has been designed and implemented to be used by the end users (e.g., marketplace users, students, etc) and run on their own mobile devices. It is important to take into consideration that end users aren't familiar with the SSI approach. Hence some concepts such as: Connections, Verifiable Credentials and Present proofs are completely new for most of them. Bearing this in mind, this mobile application has been designed and implemented to facilitate to end users the adoption of the SSI paradigm. The GUI and workflows have been adapted and simplified to improve the usability of this edge component. Figure 24 below shows the main subcomponents which are part of the mobile application.

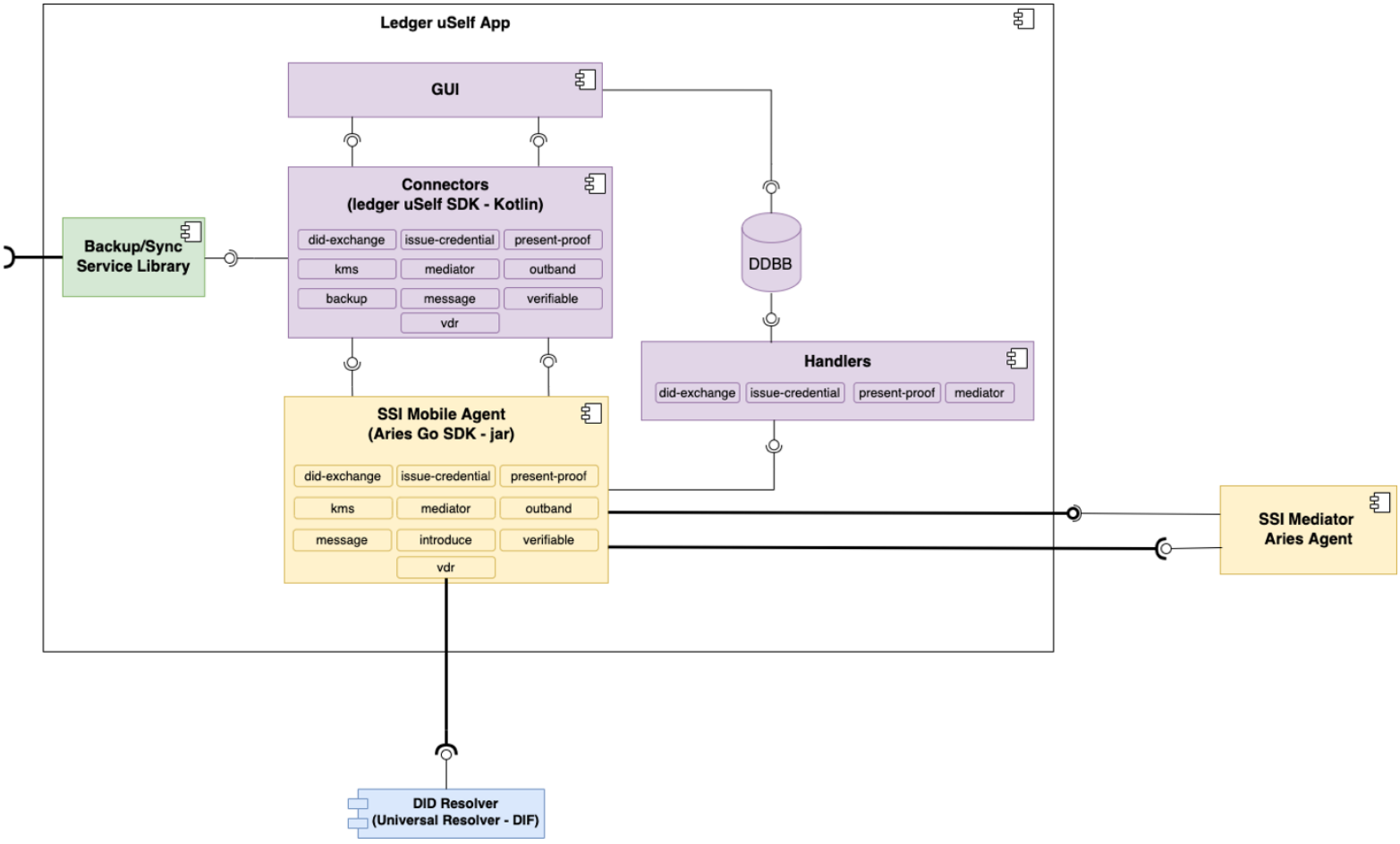

Figure 24: Mobile sub-components

The diagram above classifies the different subcomponents by colors:

- In yellow the Aries components, on one hand the mobile agent integrates its own Hyperledger Aries Agent allowing to store and manage all the identity information within the mobile. On the other hand, the mobile App requires the connection with a Mediator, that enables the communication between the other stakeholders and the mobile device.
- In blue the DID resolver, the mobile application will require access to the DID Resolver to resolve the DID documents. Inside the DID documents is the cryptographic material necessary for some of the processes performed in the mobile app.
- In green the main backup client component, which allows the end user to back up the SSI information to be restored or even to be used in different devices. Further information about the integration of this sub-component is described in the Chapter 6.
- In purple:
    - Connectors that encapsule the communication with the internal mobile agent. Further information about this component is detailed in Chapter 5.2.
    - Handlers: this set of sub-components manages the input communications received by the mobile application and triggers the corresponding GUI actions required by the end user when needed.
    - Data Base (DDBB): used to store and query the SSI's information in the mobile. This information includes the cryptographic material (i.e., the private keys) but also the connections details, the verifiable credentials, etc.
    - GUI: The Graphical User Interface set of sub-components to display a user-friendly interface for end users.

5.1.2 Interfaces

The mobile application interface has been structured in tabs to simplify different aspects of the functionality and to improve the usability.

The available tabs described in the next subsections are:

- Biometric authentication
- Connections
- Credentials
- Presentation Proof
- Backup Service

5.1.2.1 Biometric Authentication

This application contains the identity data of the end user, hence the access to it is protected by biometric authentication (fingerprint or face recognition).

Figure 25 below shows:

a) Biometric authentication, where the end user, using the corresponding sensor, authenticates to the system and
b) Authentication passed: once the end user has passed the biometric authentication, the application grants access to more sensitive features.

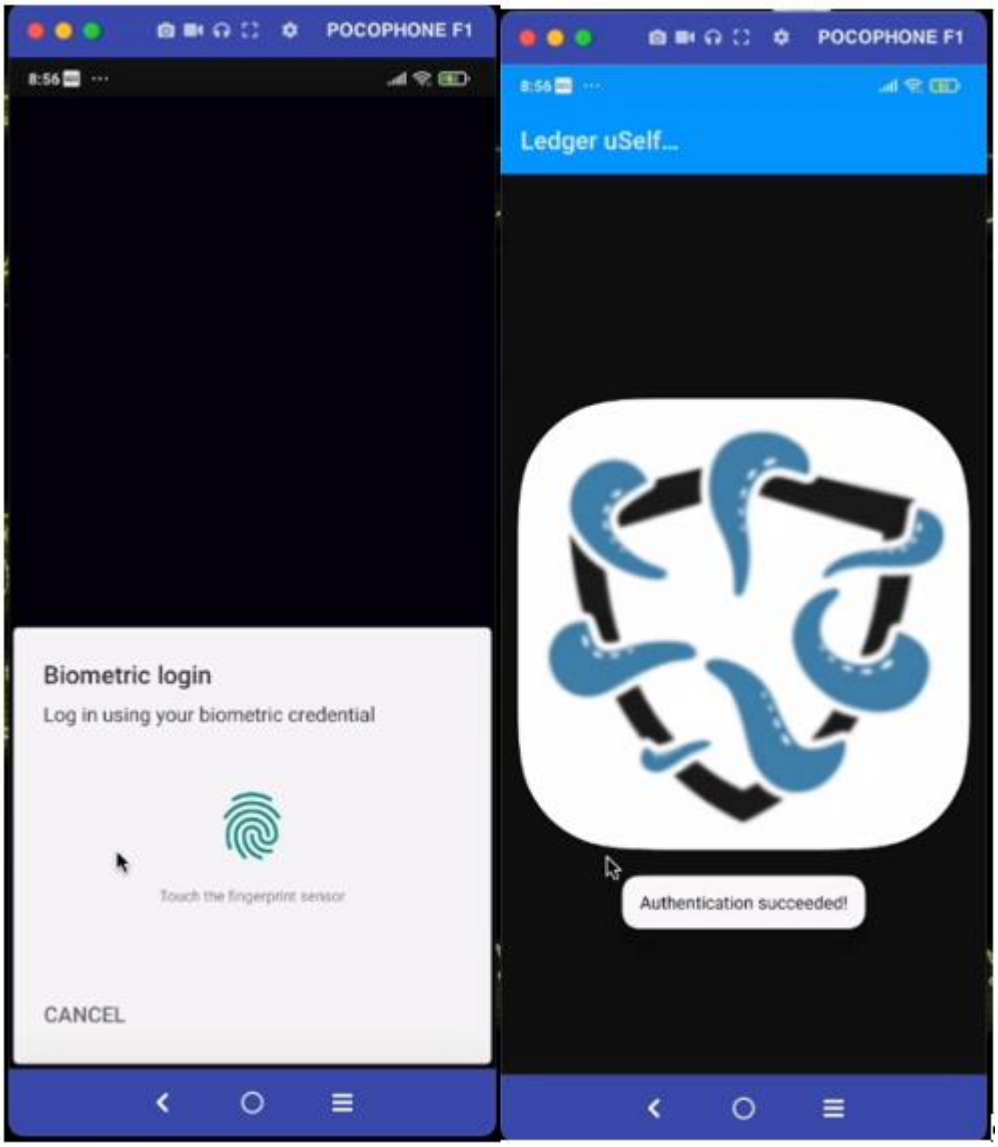

Figure 25: a) Biometric authentication challenge and b) Authentication passed

5.1.2.2 Connections

The Connections Tab shows the end user's connection details. It is worth to highlight that on the different screenshots shown in Figure 26 below, the status of the specific contact is indicated with an icon (two arrows) located in top right of the screen, in green if the connection has been completed.

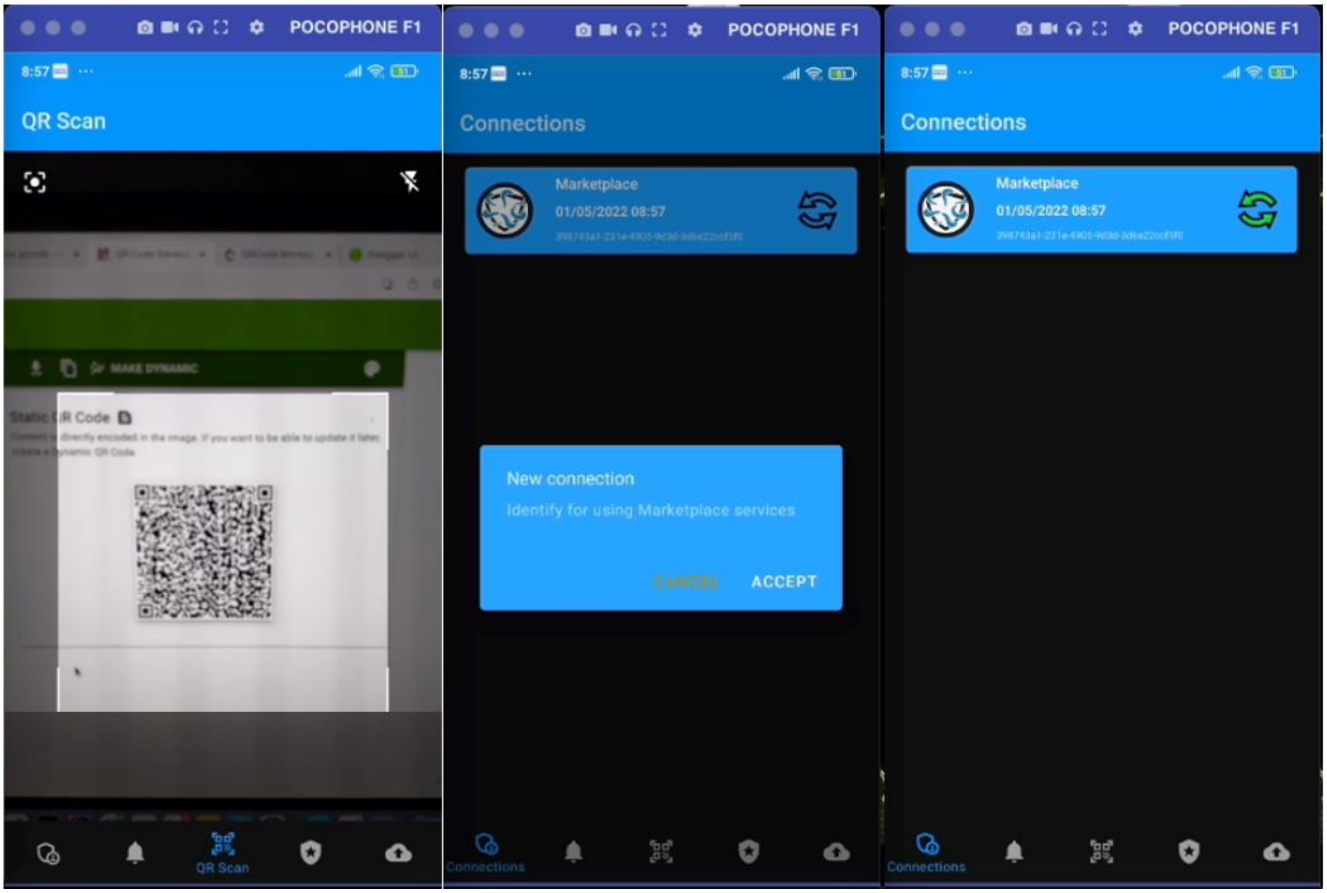

Figure 26: a) Reading QRCode Invitation, b) Accepting a new Connection and c) Connections established

Figure 26 also shows the different steps necessary to establish a connection:

a) Read the QRCode invitation that the service provider has generated to establish the connection
b) Accept the new connection if the end-user considers it appropriate and
c) The process finishes when the service provider confirms that the connection has been established and the end user can see the double green arrow on the screen

The details of the connection can be consulted in the app. See Figure 27 below.

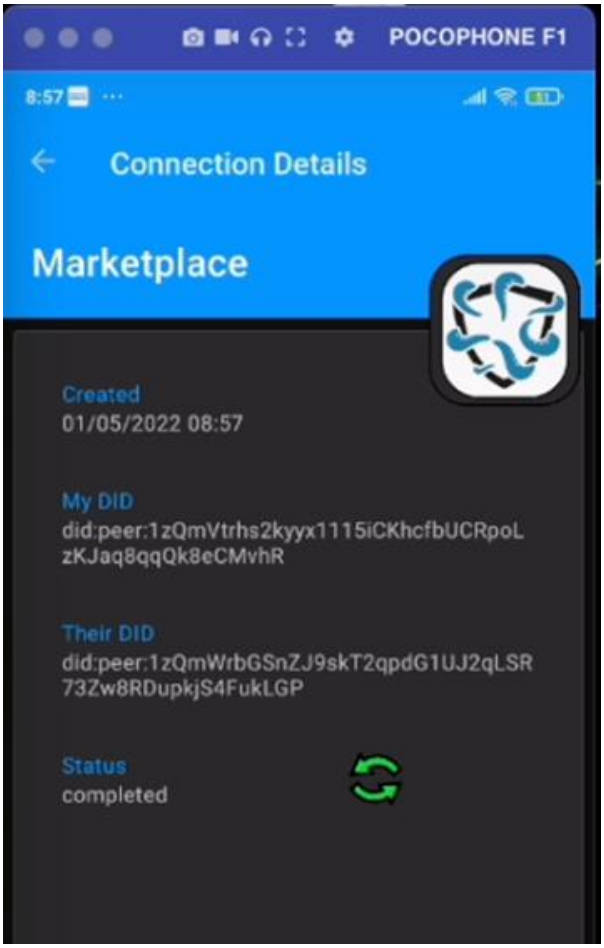

Figure 27: Connection details view

5.1.2.3 Credentials

All the information related with the obtention and the visualization of different credentials available by the end user can be accessed in the Credentials tab. The process of obtaining a verifiable credential from the issuer is performed in three different steps showed in the Figure 28 below:

a) The end user receives a credential offer sent by the service provider,
b) The end user can consult the details of the verifiable credential offered and
c) Finally, the end user receives the issued verifiable credential and stores it with the selected name.

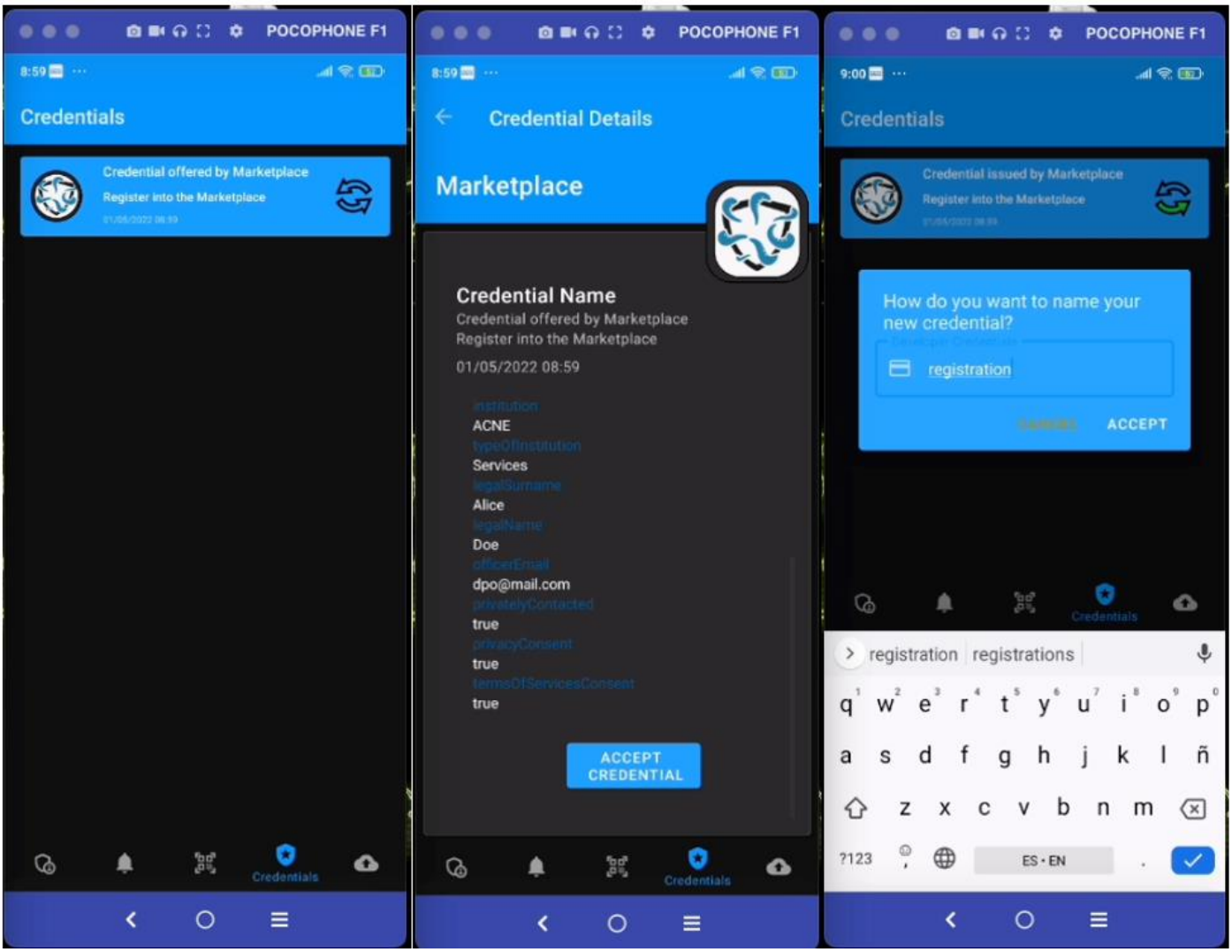

Figure 28: a) Received credential offer, b) Credential offer view and c) Store issued Verifiable Credential

The user can also consult the list of credentials stored in the app, including the raw information of the verifiable credential and its cryptographic material.

The raw verifiable credential can be shared. See Figure 29 below.

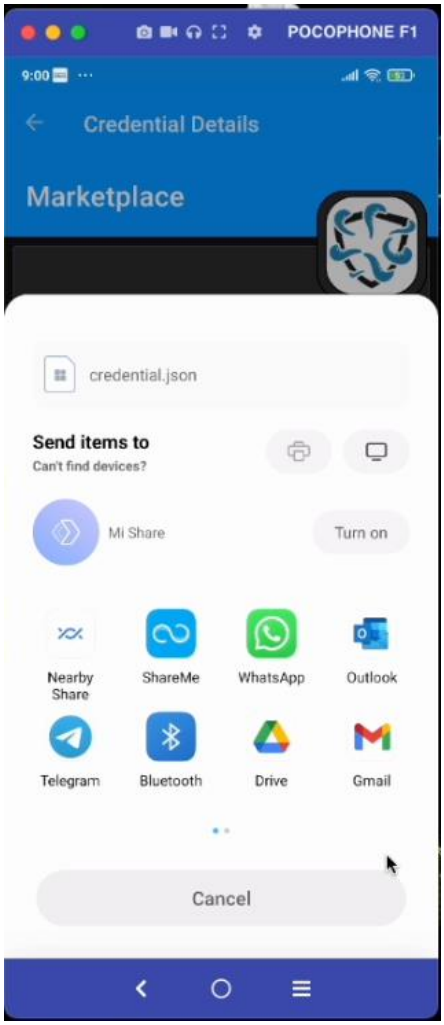

Figure 29: Share verifiable credential

5.1.2.4 Presentation Proof

To present a proof to the verifier, the mobile application receives a notification with the request. Afterwards the end user needs to select the verifiable credential that he/she wants to use for this request from the list of available and valid verifiable credentials. Finally, a presentation proof containing the information of the verifiable credential is sent to the verifier.

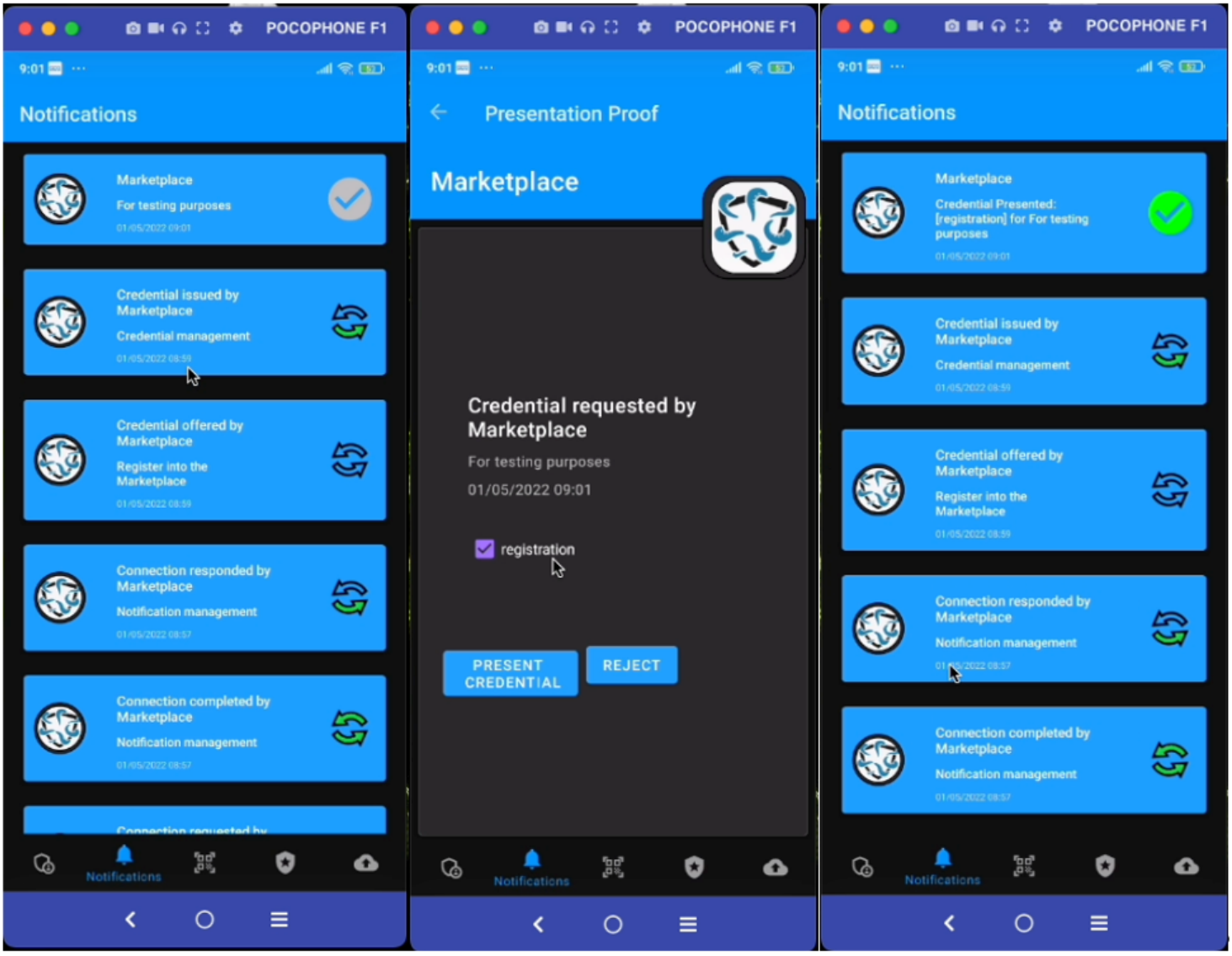

Figure 30: a) Received present proof request, b) Accept present proof and c) Present proof sent

Figure 30 shows on the left the notification of a proof request of a VC, how the end user can select the verifiable credential that suits the request and, finally, the notification that the presentation proof has been delivered.

5.1.2.5 Backup Services

Ledger uSelf mobile application allows the end user to back up his/her identity information, to restore it in a-new installation or reuse it on the installation of another device. Figure 31 shows the registration process of the mobile application. If it is the first time that the end user registers the mobile into the system, a master paper key is generated. This master paper key can be used to access to the backup information.

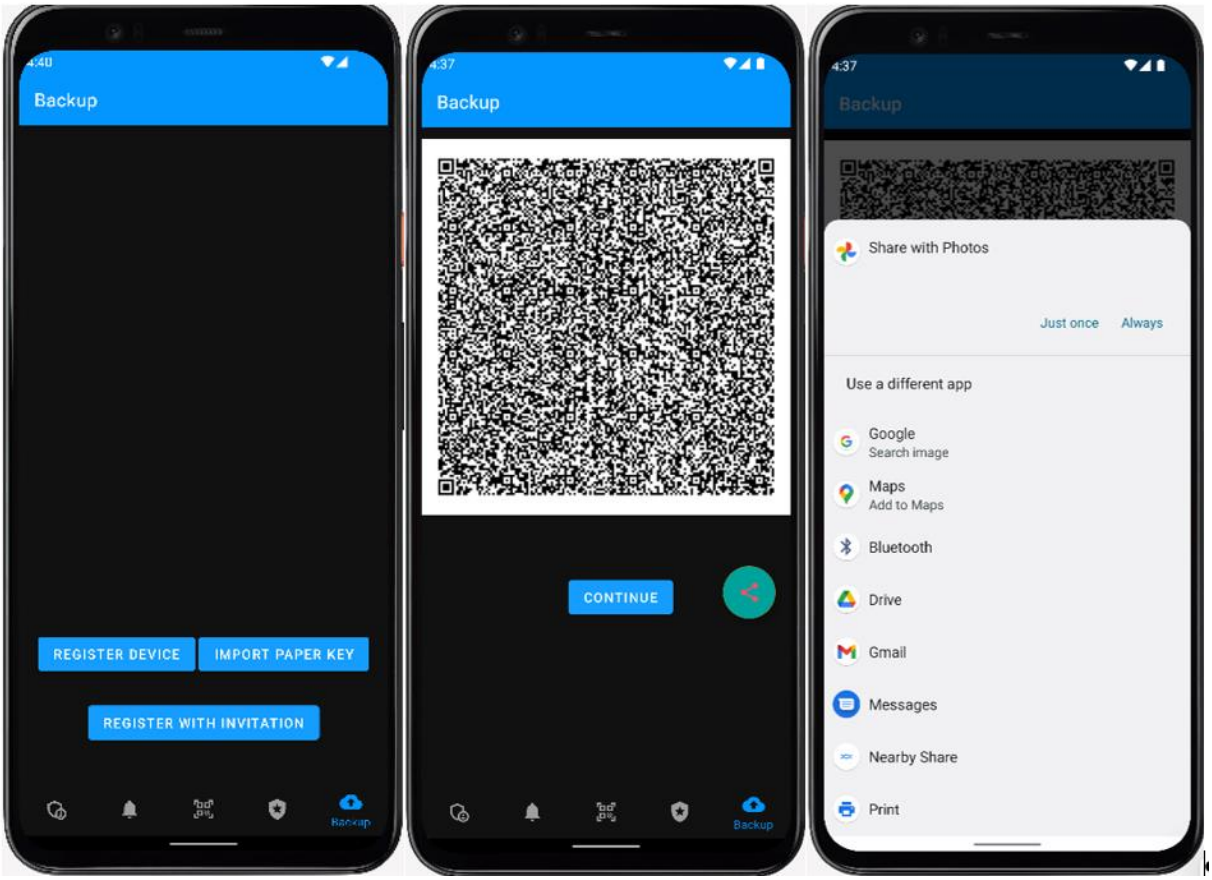

Figure 31: a) Backup service registration view, b) Master paper key generation and c) Master paper key sharing

The mobile app offers to the end user several options to store the master paper key generated by the system, for instance the end user can send the master paper key (in a QRCode format) to his/her email account.

Finally, once the end user has been registered in the backup server, he/she can back up the information or retrieve the information from the server.

5.1.3 Deployment

As described in the previous sub-sections the mobile application relies on the use of some external components. This section provides information about the deployment of the external components necessary for the proper functioning of the mobile application. The following deployment diagram Figure 32 shows these external components and their main interfaces/entry points.

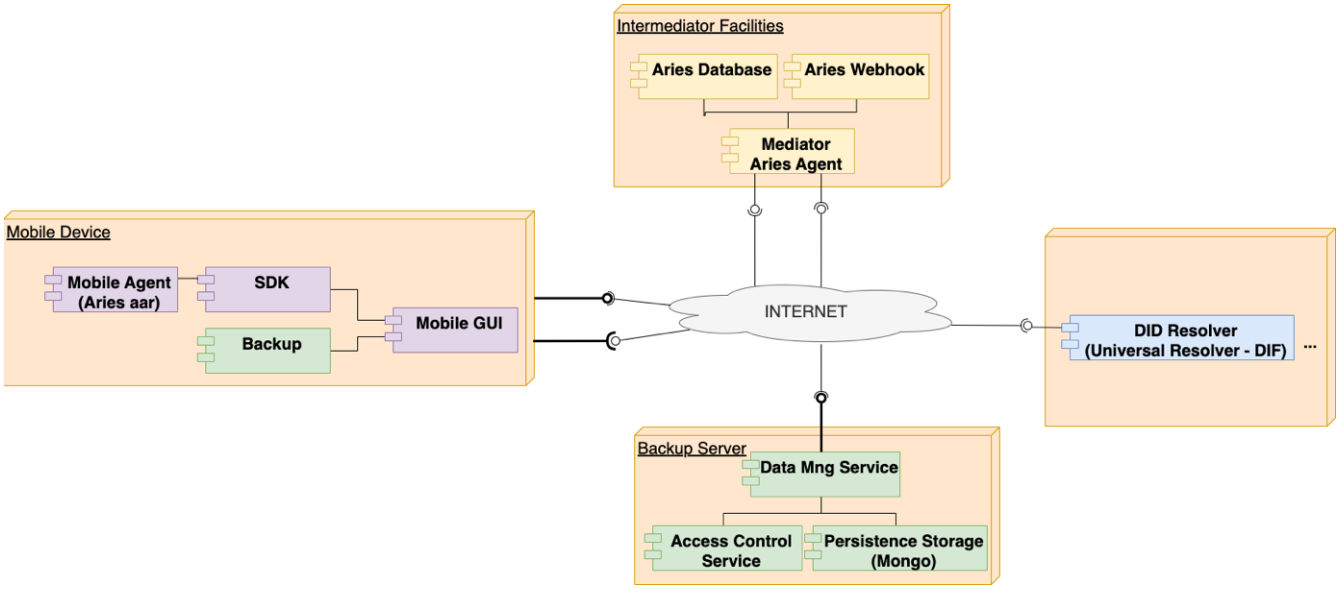

Figure 32: Mobile support components deployment diagram

Regarding the DID Resolver (in blue), it is accessed through a GET call to resolve the DID documents when necessary. With regards to the Backup Manager (further details can be seen in Chapter 6), the main entry point is the Data Management Service which provides the backup services but also the authentication and authorization of these services. Finally, the Mediator set of components, all the information of the docker images and the parameters to deploy them can be found following this link:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-testing-environment

5.1.4 Source code

The source code of the mobile application can be found following this link:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-ledgeruself-mobile

The final package application, ready to be installed on any android device, can be found following this link:

https://scm.atosresearch.eu/ari/kraken/ssi-ledgeruself-mobile-apk

5.2 Ledger uSelf SDK

The Ledger uSelf SDK is a set of sub-components that facilitates the use and hides the complexity of the direct interaction with the underlayer Hyperledger Aries Agent framework. This section describes the main changes compared to the previous version of this layer and the integration details with the mobile application.

5.2.1 Description

The available underlaying framework Hyperledger Aries Agent has been developed in Golang language. To be integrated with a mobile application the developments need to be transformed from Golang to Java. This transformation (applying gomobile software) is done by using low level primitives provided by Java Native Interface (JNI[15]) and Native Development Kit (NDK[16]) that transforms the code from

[15] https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html
[16] https://developer.android.com/ndk

Golang to C and then, from C to Java. This transformation requires that all the parameters are passed to the resulted API as bytes and byte arrays which makes the integration in the final mobile implementation much more complex. This layer carries out the conversion from normal JAVA/Kotlin object to byte arrays in both directions (from/to) the Hyperledger Ares agent in a transparent way to the developer. Figure 33 below shows the different connectors used by the mobile application.

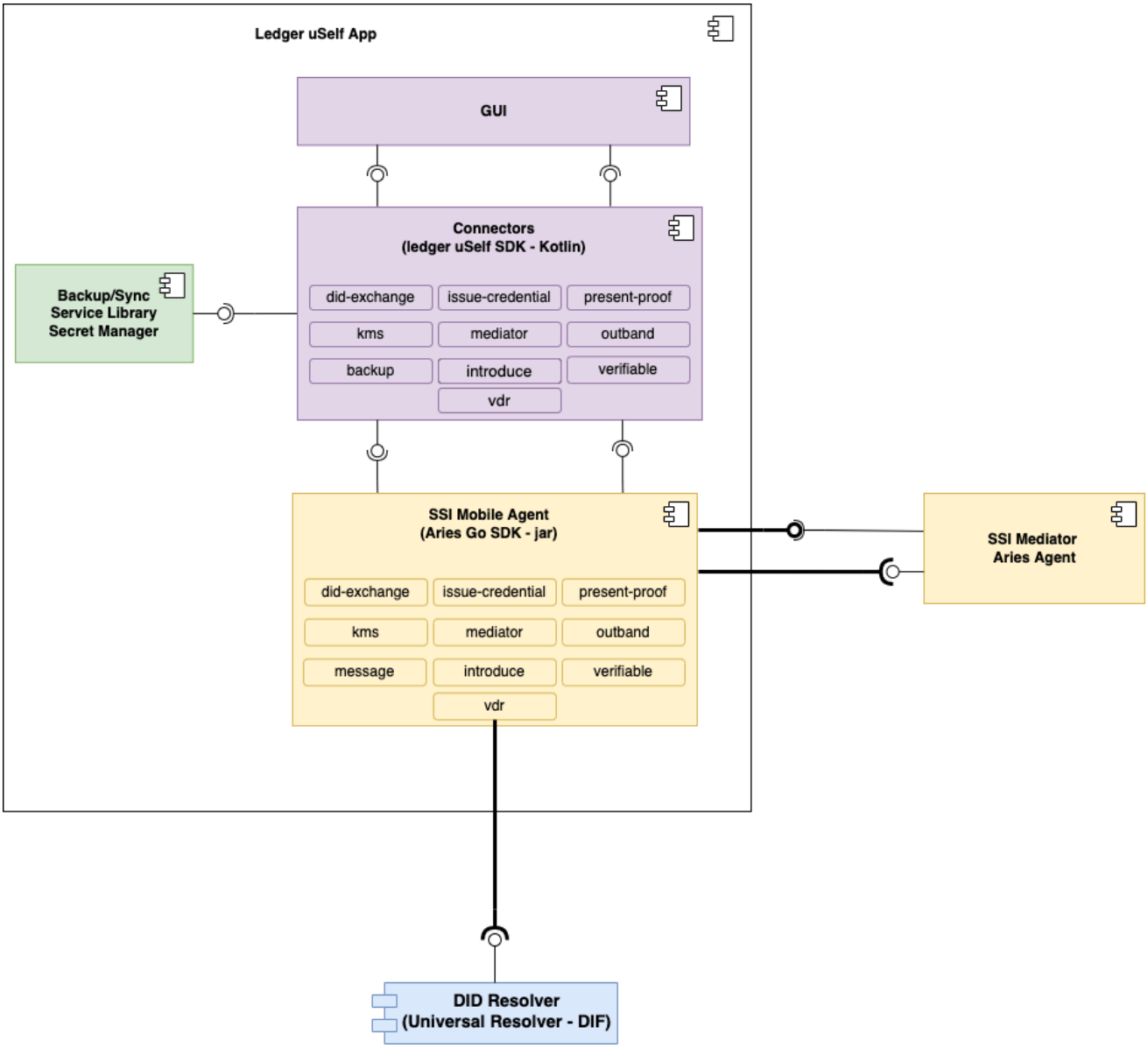

Figure 33: Ledger uSelf SDK components description

Some of these connectors were already described in the previous version of this document, however it is worth to highlight that in this version of the Ledger uSelf app a general improvement has been achieved, and new connectors have been added. For instance, the Verifiable Data Registry (VDR) (for resolving DID documents), the Key Manager Service (KMS) (for managing the cryptographic material) and the Out of Band connectors are fully implemented and integrated within this new version of this component.

5.2.2 Interfaces

The interfaces of the connectors use the interface defined by Hyperledger Aries and were described in the previous version of this report. The specifications detailing the connectors requests and parameters can be found on the following open API[17] standard file available on the following link:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-testing-environment/-/blob/main/aries/v0.1.8/all/specs/openapi-localhost-8182.yml

5.2.3 Deployment

To facilitate the integration this library has been included in the artifact repository of the KRAKEN project. It can be used just adding it as a dependency using the dependency manager selected by the final developer (Maven[18], Gradle[19], etc.). The link to the mentioned repository is:

https://registry.atosresearch.eu:8443/

5.2.4 Source code

This library is available and accessible using this link:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-ledgeruself-sdk

5.3 Ledger uSelf Broker

This edge SSI component facilitates to any Service Provider the adoption of a Self-Sovereign Identity solution, by simplifying all the process/protocols performed by this type of solution.

5.3.1 Description

As described in *D2.3 Final KRAKEN architecture* and *D2.5 KRAKEN final technical design*, some of the processes/protocols required to make use of a Self-Sovereign Identity can be complex and sometimes overwhelming for an outsider end user. Hence this component provides a simplified interface to the Service Provider. It carries out the internal processes needed to support the SSI functionality. Figure 34 below shows the different subcomponents that are part of the Ledger uSelf Broker.

[17] https://www.openapis.org
[18] https://maven.apache.org/
[19] https://gradle.org/

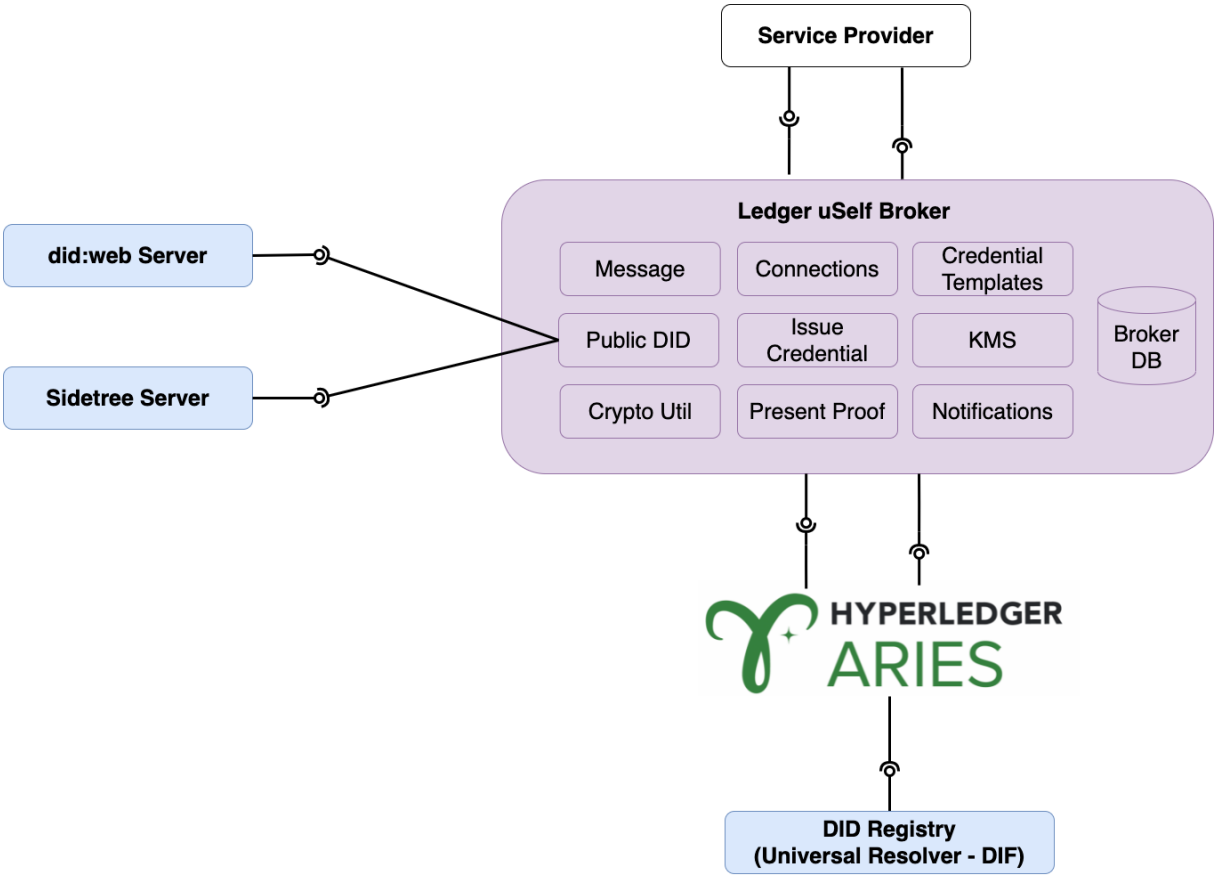

Figure 34: Ledger uSelf Broker components description

The main difference compared to the previous release of the SSI solution is that the Ledger uSelf Broker enables the Service Provider the generation of new public DIDs ready to be used by other the Service Providers. The Ledger uSelf Broker not only generates and publishes the corresponding DID document but it also creates the associated cryptographic material and imports that material within the Hyperledger Aries Agent

5.3.2 Interfaces

The previous version of this deliverable details some of the different interfaces of this component, hence the reader can refer to the already mentioned deliverable for consultation. Among the new functionalities provided by this component it is worth to highlight those associated with the management of public DIDs. See Figure 35 below:

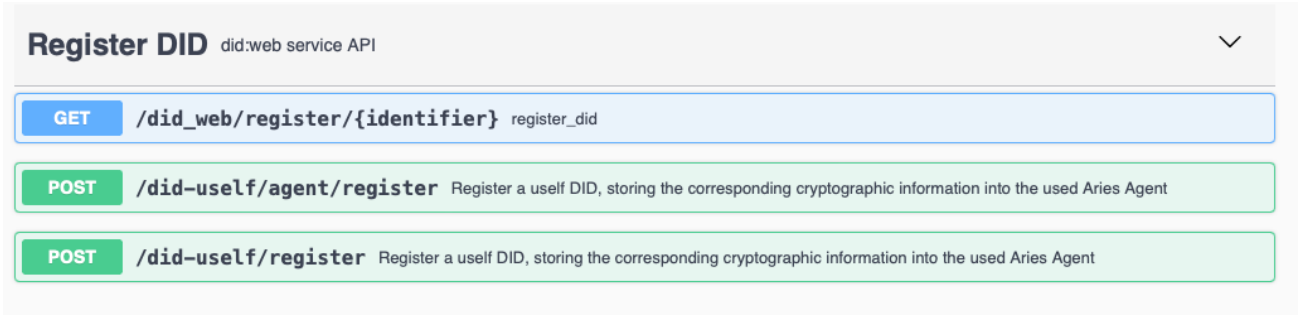

Figure 35: Ledger uSelf Broker Public DID interfaces

Where the fist interface generates a new DID document following the did:web[20] method. The last two interfaces enable publishing a DID document following a did:elem[21] method. These methods will allow the Service Provider to generate the crypto graphic material in an easy manner, import the generated keys into the agent, generate the associated DID document and then, publish the new DID document in the corresponding registry.

5.3.3 Deployment

To use this component, it is necessary to deploy some supporting subcomponents. The following diagram (Figure 36) shows the different subcomponents and their interfaces and iterations.

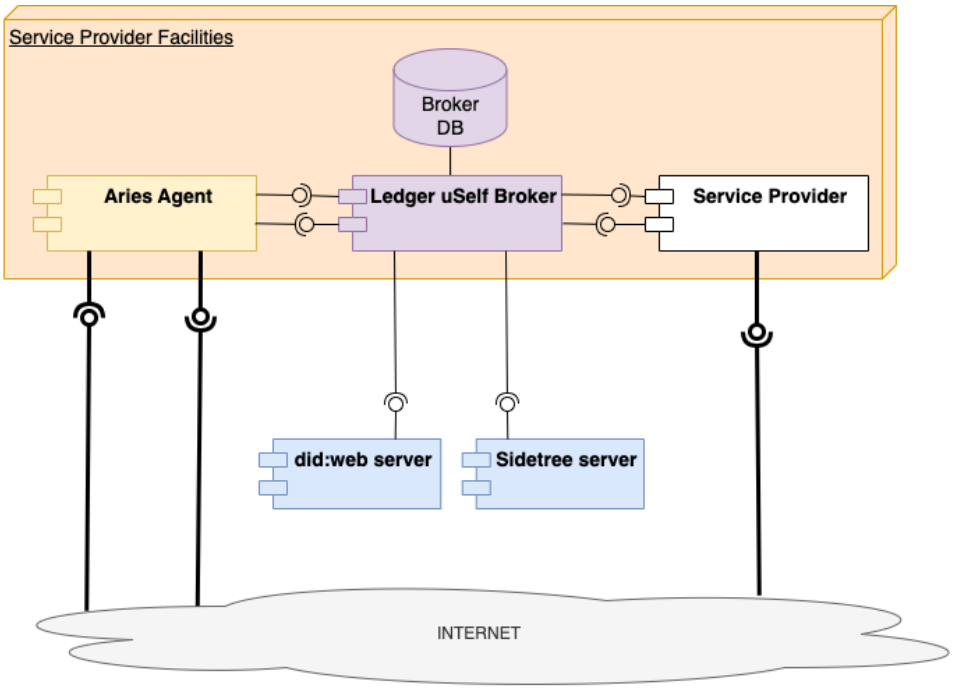

Figure 36: Ledger uSelf Broker components deployment

These components must be deployed within the Service Provider facilities as they will contain its Self-Sovereign Identity information (including the public and private keys). In addition, as the reader can see on the diagram the Ledger uSelf broker is a component that can only be accessed from the internal net of the Service Provider facilities and not from outside.

5.3.4 Source code

The source code of the Ledger uSelf Broker can be accessed using the following link:

https://scm.atosresearch.eu/ari/ledger_uself/ssi-ledgeruself-broker

[20] https://github.com/w3c-ccg/did-method-web
[21] https://github.com/decentralized-identity/element/blob/master/docs/did-method-spec/spec.md

6 T3.4 SSI Wallet Management

The design of SSI Wallet Management component and main modules has not changed since the delivery of the previous iteration of this document (*D3.1 Self-Sovereign Identity Solution First Release* [4]). Therefore, the content of this section reflects the content from D3.1 in large parts and was updated where necessary.

The SSI Wallet Management is carried out by two components: The Secret Manager (Backup/Sync Library) and the External Remote Storage (Backup/Synch Server). The External Remote Storage runs on a remote server and hosts encrypted backups, while the secret manager library is integrated into a mobile application where it enables the access to the services of the External Remote Storage. **The goal of both components is to keep the user's secrets (e.g., verifiable credentials, cryptographic material) safe (confidentiality) and available.** The communication between secret manager and external remote storage takes place via the DID Exchange and the DID Auth Protocol. To facilitate the communication, the components implement and run relevant parts of these protocols. Figure 37 depicts the components developed for the SSI Wallet management on both the client and the server side.

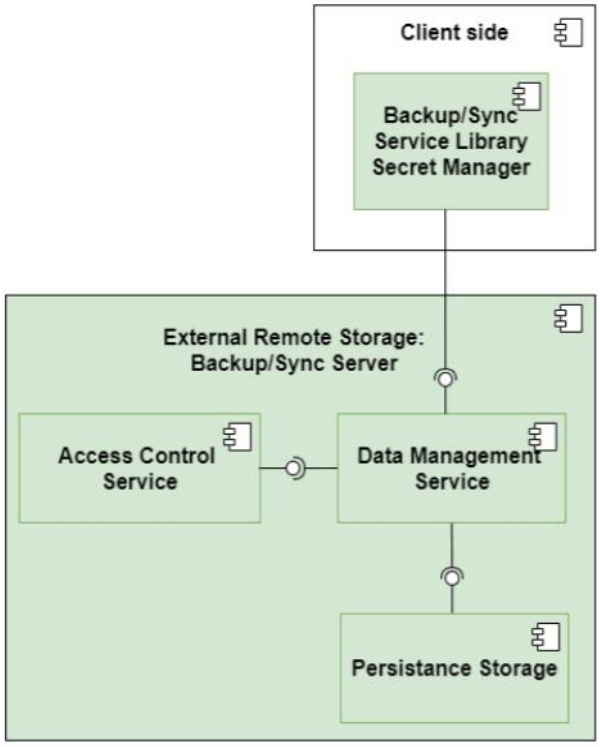

Figure 37: SSI Wallet Management components design

6.1 External Remote Storage: Back-up Synch Server

6.1.1 Description

The External Remote Storage (ERS) offers users to store and synchronize secrets across multiple devices. These secrets are encrypted from end-to-end, meaning that the data is being encrypted and decrypted on the user's device and only shared with the ERS in an encrypted form. Using proxy re-encryption, the ERS can aid the synchronization of secrets between different devices without having access to any private encryption key **or to the secret**.

The ERS is implemented by the following three applications:

- The **Data Management Service (DMS)** is a web application that stores, retrieves and re-encrypts secrets.
- The **Access Control Service (ACS)** handles registration and authentication of devices.
- The **Persistent Storage** stores user secrets in a persistent storage.

6.1.2 Interfaces

A client interacts with the External Remote Storage via the following RESTful HTTPS endpoints: The Data Manager Service endpoint at /dms and the Access Control Service endpoint at /did. The following subsections enumerates and describes all operations that the services offer.

6.1.2.1 Access Control Service

The Access Control Service (ACS) offers a public API via the /acs endpoint. A client uses the operations offered on this endpoint to register and authenticate a *pairwise peer DID*[22]. The sequence diagram in Figure 38 shows how a client device registers its DID. Note that the exchange invitation can come from the ACS (in case when users register their first device) or from an already registered device (in case when users register multiple devices that should be synchronized) [4].

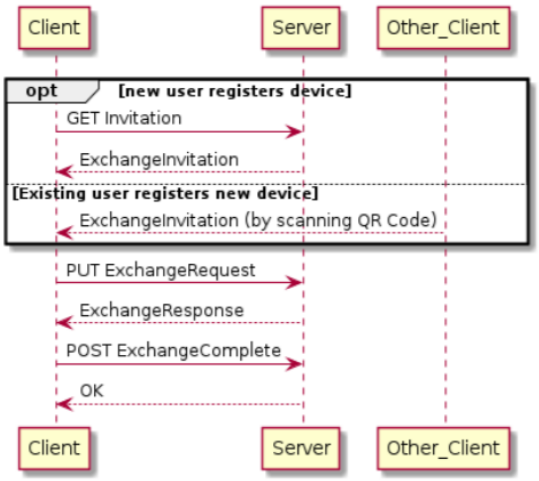

Figure 38: Sequence Diagram of Client and External Remote Storage While performing the DID Exchange protocol [4]

The registration follows the DID Exchange Specification [9] using the following operations:

- GET /did/exchange: Request invitation.
  - Header: none
  - Query: none
  - Payload: none
  - Response: QR image with ExchangeInvitation Message
- GET /did/exchange/json: Request an implicit invitation, wrapped in a json message structure.
  - Header: none
  - Query: none
  - Payload: none

[22] Peer DID Method Specification, https:/identity.foundation/peer-did-method-spec/ (on 2021-06-22)

    - Response: JSON with QR image of ExchangeInvitation Message
- PUT /did/exchange/request: Send an exchange request.
    - Header: none
    - Query: none
    - Payload: ExchangeRequest Message
    - Response: ExchangeResponse Message
- POST /did/exchange/complete: Finalize DID exchange.
    - Header: none
    - Query: none
    - Payload: ExchangeComplete Message
    - Response: 200 (OK) or 400 (BAD_REQUEST)

After the DID of the device has been exchanged, the client proofs ownership of the DID's private key using the DID Auth protocol[23] (Figure 39)

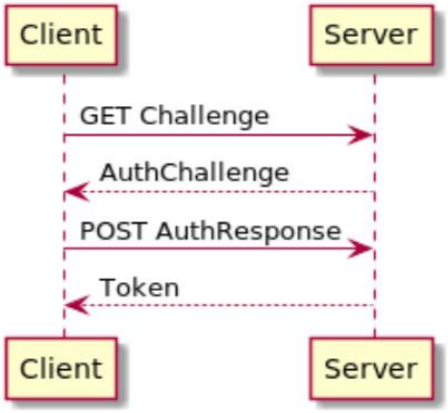

Figure 39: Sequence Diagram of Client that proofs ownership of DID towards Server via DID Auth [4]

The ACS offers the following operations to carry out the DID Auth protocol:

- GET /did/auth: get authentication challenge
    - Header: did
    - Query: -
    - Payload: none
    - Response: AuthChallenge Message
- POST /did/auth: send response of authentication challenge, get DIDAuthToken
    - Header: did
    - Query: -
    - Payload: AuthResponse Message
    - Response: DIDAuth JWS Token

6.1.2.2 Data Management Service

The data management service offers its functionality via a private API. Requests to this API need to be authenticated via the DIDAuth JWS Token from the DID Auth Protocol. Requests need to be authenticated by putting the DIDAuthToken in the HTTP Request Authorization Header, as follows:

[23] DID Auth, https://github.com/WebOfTrustInfo/rwot6-santabarbara/blob/master/final-documents/did-auth.md (on 2021-06-22)

GET /dms/register-user HTTP/1.1

HOST ssi-external-remote-storage.iaik.tugraz.at

Authorization: DIDAuthToken <token>

The following listing describes the operations that the DMS offers:

- POST /dms/register-device: This endpoint adds the device to a group of devices that share secrets. The endpoint can only be used after the client device was invited from another device. In its response it hands out all public keys of other devices of the same group. The client is expected to issue and upload re-encryption keys for these devices.
    - Header: Authorization with DidAuthToken
    - Query: none
    - Payload: none
    - Response: UserIdGroup message, which contains the public keys of other devices.
- PUT /dms/reks: Upload a re-encryption key. Necessary for the ERS to be able to re-encrypt data from one device to another device.
    - Header: Authorization with DidAuthToken
    - Query: none
    - Payload: {rencryptionkeyB64, {ownerId, recvrId}}
    - Response: 200 (OK), 400 (BAD_REQUEST), 409 (CONFLICT)
- GET /dms/synchronize: This endpoint is similar to /dms/register-device, but it needs to be invoked by devices that are already registered. The response contains public keys of devices in the same group for which the ERS does not have a re-encryption key. Usually, these other devices joined the group after the client's device. The client is expected to issue and upload re-encryption keys for these devices.
    - Header: Authorization with DidAuthToken
    - Query: none
    - Payload: userIdGroup
    - Response: UserIdGroup message, which contains the public keys of other devices.
- GET /dms/data-root/{userId}: Get a user's root folder. This is the user's root directory for storing files and subdirectories.
    - Header: Authorization with DidAuthToken
    - Query: userId of owner's data root
    - Payload: none
    - Response: Metadata object that describes data root, including its dataId.
- GET /dms/data/{dataId}/children: List files or directories that belong to the folder identified by dataId.
    - Header: Authorization with DidAuthToken
    - Query:
        - dataId, identifies a folder in the ERS.
        - dataType: either 'file' or 'dir'. If you want to list files or folders that belong to the folder identified by dataId.
        - startIndex: optional integer. Return only children with an index greater or equal this number.
        - endIndex: optional integer. Return only children with an index less or equal this number.

    - Payload: none
    - Response: List of Metadata of Children
- GET /dms/data/{dataId}: Download the content of a file.
    - Header: Authorization with DidAuthToken
    - Query: dataId, identifies the file.
    - Payload: none
    - Response: metadata and reencrypted file contents, needs to be decrypted by client device.
- PUT /dms/data/{dataId}: Upload a file
    - Header: Authorization with DidAuthToken
    - Query: dataId, identifies the file.
    - Payload: Create Request Message, including metadata and fileContent.
    - Response: 201 (CREATED), 400 (BAD REQUEST), or 500 (INTERNAL SERVER ERROR)
- POST /dms/deregister-user/{userId}: Delete the user and her data.
    - Header: Authorization with DidAuthToken
    - Query: userId, identifies the user
    - Payload: none
    - Response: 200 (OK), 404 (NOT FOUND), 400 (BAD REQUEST)

6.1.3 Deployment

The ERS will be deployed at a host in the TU Graz network and will be reachable through the following URL:

https://kraken-ssi-external-remote-storage.iaik.tugraz.at/

Both the Access Control Service and the Data Management Service are web applications that will be deployed in a tomcat container app. While these apps are reachable through the public network, the persistent storage which runs as MongoDB instance on the same host is not reachable through the public network.

6.1.4 Source code

The source code of the External Remote Storage is available on the KRAKEN project GitHub at:

https://github.com/krakenh2020/ssi-backup-library/tree/master/ssi-backup-library-server

6.2 Secret Manager: Back-up Synch Service Library

6.2.1 Description

The Secret Manager is a library that connects a client application with the External Remote Storage. The tasks of the Secret Manager can be divided into two categories: On the one hand, the Secret Manager handles the interactions with the External Remote Storage, such as registering, authenticating towards the Access Control Service, as well as storing and retrieving secrets from and to the Data Management Service. On the other hand, the secret manager performs all client site cryptographic operations (encrypting, de-crypting secrets, issuing re-encryption keys) and manages the associated key material. The secret manager offers its functionality via an easy-to-use API that relieves a client application from dealing with the complexity of managing key material and interacting with the ERS.

6.2.2 Interfaces

The API that the Secret Manager exposes towards a client application consists of the following operations:

- void init(Context context)
    - This method initializes the components in the secrets manager library.
    - The method must be called before performing any other interaction with the library.
    - If not available, the method generates re-encryption key pair and stores it locally.
    - If not available, the method generates client backup authentication key pair.
    - If not available, the method creates a DID from backup authn key pair for communication with External Remote Storage.
    - The context parameter enables access to the file system to store and retrieve data.
- Bitmap generateExchangeInvitation()
    - This method generates an Exchange Invitation that is encoded in a QR code bitmap.
    - If scanned by a new device, the QR Code enables access to the group of devices that share secrets.
- Result registerDevice (String deviceName)
    - This method registers the current device at the ERS and associates the device with a new user.
    - Call this method if you want to register a new device without access to any previously stored secrets on the default ERS.
    - The operation returns a Result which tells if the registration succeeded, and, in case it failed, a message explaining the reason.
- Result registerDevice (String exchangeInvitation, String deviceName)
    - This method registers this device at the ERS and associates the device with the user who issued the invitation.
    - Call this method if you want to register a new device to an existing user and share secrets in between devices, or if you simply want to register at a service other than the default ERS.
    - The exchangeInvitation Parameter is a JSON Web Signature Token that the client application obtains from the ERS or from another device, typically by scanning a QR code.
    - The operation returns a Result which tells if the registration succeeded, and, in case it failed, a message explaining the reason.
- void synchronizeKeys()
    - Ensures that the ERS receives the key material for re-encrypting data between devices.
    - Retrieves public re-encryption keys from other devices in the same group.
    - Creates re-encryption keys for each public re-encryption key on the device.
    - Uploads these re-encryption keys to the ERS.
- Result backupSecret(String id, String secret)
    - This method encrypts and stores a secret on the ERS.
    - The id parameter identifies the secret and is used for retrieval. This parameter is sent to the ERS in plain.
    - The secret paramater contains the secret in plaintext. This method encrypts the secret with the private key of the on-device re-encryption key pair. Afterwards, the resulting ciphertext of the secret is sent to the ERS.

- The operation returns a Result which tells if the backup succeeded, and, in case it failed, a message explaining the reason.

- Map<String, Optional<String>> restoreSecrets()
  - Retrieves and decrypts all secrets from the ERS.
  - This method returns a map that is indexed by the secret id.
  - If a secret could not be re-encrypted for the device due to a non-existent re-encryption key, an empty Optional value is stored in the map for its identifier.
- Optional<String> restoreSecret(String id)
  - Retrieves a single re-encrypted secret from the ERS. The value is then decrypted with the device's private re-encryption key.
  - The parameter id identifies the secret.
  - This method returns the decrypted secret in plain (if available) or an empty Optional if the secret could not be re-encrypted for the device due to a missing re-encryption key.
- List<String> listSecrets()
  - This method retrieves all secrets-identifiers for the user.
  - The identifiers can be used to selectively retrieve single secret-values from the server.
- Bitmap exportPaperkey()
  - Generates a paperkey which is used to restore all secrets.
  - The paperkey contains private key information and must be stored confidentially.
  - It is recommended to include the bitmap in a PDF file, which is then printed and stored in a secure location.
  - This method returns a QR Code bitmap that contains the paperkey information.
- Result importPaperkey(String paperkey)
  - Uses the value stored in the paperkey QR code to restore information when all devices are lost.
  - Some public information is retrieved from the server before the device can be restored.
  - All information previously stored on this device is replaced after calling paperkey!
  - The paperkey parameter contains a JSON Token that needs to be parsed from the paperkey QR code.
  - This method returns success or error with an error message.

6.2.3 Deployment

The Secret Manager Backup/Sync Library is deployed as a maven artifact to the ATOS KRAKEN Nexus repository.

To use the library as an artifact in a maven project, copy the following snippet into the project/dependencies section of your pom.xml:

```
<dependency>
  <groupId>eu.krakenh2020</groupId>
  <artifactId>ssi-backup-library-client</artifactId>
  <version>$version</version>
</dependency>
```

In case of using the secret manager in a gradle project, add the following line to your build.gradle file:

```
implementation 'eu.krakenh2020:ssi-backup-library-client:$version'
```

6.2.4 Source code

The source code of the External Remote Storage is available on KRAKEN project GitHub at:

https://github.com/krakenh2020/ssi-backup-library/tree/master/ssi-backup-library-client

6.2.5 Baseline technologies and tools

The Secret Manager Backup / Sync Library is written in Java 8. Like the External Remote Storage, the secret manager relies on cryptographic primitives from the IAIK JCE[24] and parses / serializes JSON with FasterXML Jackson Core library[25]. The zxing[26] image processing library generates and parses the QR Code of the Exchange Invitation.

[24] https://jce.iaik.tugraz.at/
[25] https://github.com/FasterXML/jackson-core
[26] https://github.com/zxing/zxing

7 Conclusion

This document is a report of the work performed in tasks T3.1, T3.2, T3.3 and T3.4 of the KRAKEN project. The activity has focused on the development of a prototype of the project SSI solution, one of the three main pillars of the KRAKEN platform. The components supporting the SSI functionalities needed by the pilots' service providers (the marketplace and the university) have been implemented. The SSI solution is based on the results of WP2 Technical aspects and architecture specifications, specifically in *D2.3 Final KRAKEN architecture* [8] and *D2.5 KRAKEN final technical design* [7]. This report builds on the first version of the SSI solution *D3.1 Self-Sovereign Identity Solution First Release* [4], result of the first development cycle, the improvements and the new functionalities developed during the second iteration, such as the KTIR, the KTRER and the DID resolution.

The SSI components will be integrated with the KRAKEN marketplace and validated in the project pilots, the health and education pilots. The assets produced in T3.1, the LIM and the KWCT, are used to generate and manage verifiable credentials derived from trustable entities such as the eIDAS network. The results of T3.2 include the Trusted Framework, which facilitates the integration of the SSI solution with the KRAKEN marketplace by decoupling and abstracting from the blockchain complexity the KRAKEN data sharing process. Additionally, T3.2 facilitates the creation of a scenario aligned with EBSI/ESSIF. The Ledger uSelf edge tools are the outcomes of T3.3 and are based on the results of T3.2 and T3.4. These tools are the Ledger uSelf app for managing the VC and key material, the Ledger uSelf SDK for a more friendly use of the Hyperledger Aries framework and the Ledger uSelf broker for integrating the SPs, facilitate the adoption of SSI solution on multiple devices.

Future improvements could include new methods for DID resolution, and a tighter integration with the EBSI/ESSIF building blocks currently released by the EC, thanks to the modular design of the provided KRAKEN SSI solution.

Finally, in the scope of T3.4 the External Remote Storage and the Secret Manager for handling backup and synchronization of the SSI Wallet, useful when the mobile device is lost or stolen, were developed. The Secret Manager encrypts and decrypts data, whereas the remote storage handles re-encryption between sender and recipient. To perform this re-encryption, the remote storage needs access to the re-encryption key.

Currently, the process of issuing and uploading the re-encryption key requires the interaction of the sender. For future improvements, the synchronization of re-encryption keys could be implemented in a non-interactive protocol, e.g., by utilizing mobile push notifications.

8 References

[1] Decentralized Identity Foundation DIF. https://identity.foundation

[2] Hyperledger Aries Framework Go. https://github.com/hyperledger/aries-framework-go

[3] European Commission. European Blockchain Service Infrastructure EBSI. https://ec.europa.eu/digital-building-blocks/wikis/display/EBSI/Home

[4] KRAKEN project: D3.1 Self-Sovereign Identity Solution First Release, Palomares A., 2020. https://www.krakenh2020.eu/sites/kraken/files/public/content-files/deliverables/KRAKEN_D3.1%20Self%20Sovereign%20Identity%20Solution%20First%20Release_v1.0.pdf

[5] High-Level Architectural Blueprint of ESSIFv2. https://ec.europa.eu/digital-building-blocks/wikis/pages/viewpage.action?pageId=380862977

[6] Architecture Diagram of EBSI V2. https://ec.europa.eu/digital-building-blocks/wikis/display/EBSIDOC/Architecture

[7] KRAKEN project: D2.5 KRAKEN final technical design, Palomares A., 2021.

[8] KRAKEN project: D2.3 Final KRAKEN architecture, Porro D., 2021.

[9] Hyperledger Aries RFC 0023: DID Exchange Protocol 1.0. https://github.com/hyperledger/aries-rfcs/tree/main/features/0023-did-exchange

[10] KRAKEN project: D2.4 KRAKEN Intermediate Technical Design, Palomares A., 2020.

[11]Hyperledger Aries RFC 0453: Issue Credential Protocol 2.0. https://github.com/hyperledger/aries-rfcs/tree/main/features/0453-issue-credential-v2

[12] Hyperledger Aries RFC 0454, Present Proof Protocol 2.0. https://github.com/hyperledger/aries-rfcs/tree/main/features/0454-present-proof-v2

[13] JSON Scheme. https://json-schema.org/draft-07/json-schema-release-notes.html

[14]https://www.keycloak.org/

[15]https://openid.net/connect/

[16] https://github.com/italia/spid-go

[17] https://www.w3.org/TR/vc-data-model/

[18]https://github.com/hyperledger/aries-cloudagent-python/blob/main/docs/GettingStartedAriesDev/AriesAgentArchitecture.md

[19]https://github.com/hyperledger/aries-cloudagent-python

@KrakenH2020

Kraken H2020

www.krakenh2020.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871473