

University of Padua

DEPARTMENT OF MATHEMATICS "TULLIO LEVI-CIVITA"

BACHELOR'S DEGREE IN COMPUTER SCIENCE



Owning your data through Self-Sovereign
Identity: agents implementation for Verifiable
Credentials interaction

Bachelor thesis

Supervisor

Prof. Alessandro Brighente

Co-Supervisors

Prof. Mauro Conti

Dott. Mattia Zago

Graduating

Matteo Casonato

ACADEMIC YEAR 2021-2022

Abstract

Nowadays, most of our data is owned by private companies, and everyone knows everything about us because privacy online is not well preserved. Imagining a world different from this is difficult, but things can change thanks to Self-Sovereign Identity (SSI). SSI approach aims to bring credentials back to the actual owners, the people. This is possible through cryptography and secure authentication layers (e.g., OAuth, OpenIDConnect). The developed product embraces this philosophy and offers a solution where the users are the holders, issuers, or verifiers of Verifiable Credentials (VCs). Specifically, will be developed software agents who create, issue, verify, modify or even revoke the credentials, leveraging an SSI Kit.

In this thesis, we propose a methodology to merge SSI off-chain (i.e., outside the blockchain) operations with on-chain smart contracts. In particular, the job has been divided into three macro stages: firstly, has been done a deep dive into the SSI technology, studying all of its primitives and analyzing the problem; secondly, has been developed a Software Development Kit (SDK), which enabled us to dialog with an SSI Kit (off-chain logic); in the meantime, my friend and co-worker Matteo Midena developed the smart contracts (on-chain logic); finally, off-chain and on-chain solutions has been merged in a proof of concept web application.

*“If you always do what you’ve always done,
you’ll always get what you’ve always got.”*

— Henry Ford

Acknowledgments

First of all, I would like to thank the people who helped me during the writing of this paper: my supervisor Prof. Alessandro Brighente and my co-supervisors Prof. Mauro Conti and Dott. Mattia Zago.

Also, I want to thank my parents, who have always supported me, and never stopped me from doing anything I truly wanted.

Finally, I thank my friends, who have eased these sometimes intense but very satisfying years.

Padova, September 2022

Matteo Casonato

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | The problem | 1 |
| 1.2 | Basic use cases | 3 |
| 1.3 | Internship description | 5 |
| 1.3.1 | The company | 5 |
| | Athesys | 5 |
| 1.3.2 | Internship objectives and planning | 5 |
| 2 | State of the art and technology background | 7 |
| 2.1 | Technology concepts | 7 |
| 2.1.1 | Self-Sovereign Identity concepts | 7 |
| | Self-Sovereign Identity | 7 |
| | Verifiable Credential (VC) | 8 |
| | Verifiable Presentation (VP) | 9 |
| | Decentralized Identifier (DID) | 10 |
| | JavaScript Object Notation (JSON) | 11 |
| 2.1.2 | Blockchain concepts | 11 |
| | Blockchain | 11 |
| | Permissionless and permissioned blockchains | 12 |
| | Ethereum | 12 |
| | Hyperledger | 13 |
| | Hyperledger Besu | 13 |
| | Hyperledger Fabric | 14 |
| 2.1.3 | Libraries and Stack involved | 14 |
| | EBSI | 14 |
| | walt.id SSI Kit | 15 |
| 2.2 | State of the art | 15 |
| 2.2.1 | Complete solution | 15 |
| 2.2.2 | SSI Kits | 15 |
| 2.2.3 | Smart contract suites | 17 |
| 3 | Solution | 18 |
| 3.1 | Solution proposal | 18 |
| 3.2 | Solution development | 20 |
| 3.2.1 | Technologies and Tools | 20 |
| | Common tools and languages | 20 |
| | SSI Kit SDK | 20 |
| | Frontend | 21 |

| | | |
|-------|--|-----------|
| | Backend | 21 |
| 3.2.2 | SSI Kit SDK development | 21 |
| | walt.id SSI Kit | 21 |
| | SSI Kit Typescript SDK | 22 |
| 3.2.3 | Web Application Proof of Concept | 31 |
| | Structure | 31 |
| | Functionalities | 33 |
| | Problems and difficulties found | 38 |
| 3.3 | Discussion | 38 |
| 3.3.1 | Achievements | 38 |
| 3.3.2 | Acquired knowledge | 38 |
| 3.3.3 | Future developments | 38 |
| 3.3.4 | Personal evaluation | 38 |
| | Bibliography | 40 |

List of Figures

| | | |
|------|--|----|
| 1.1 | University canteen use case | 4 |
| 1.2 | Athesys logo | 5 |
| 1.3 | Monokee logo | 5 |
| 1.4 | Internship structure | 6 |
| 2.1 | The triangle of trust: Prover, Issuer, and Verifier (by Tykn) | 8 |
| 2.2 | Example of verifiable credential (VC) | 8 |
| 2.3 | Example of verifiable presentation (VP) | 9 |
| 2.4 | Example of a DID | 10 |
| 2.5 | DID architecture overview and basic components relationship | 10 |
| 2.6 | Example of DID document | 10 |
| 2.7 | Simple blockchain visualization | 11 |
| 2.8 | Only allowed users can participate in the network | 13 |
| 2.9 | Mary cannot see the private transaction sent from Alice to Bob | 13 |
| 2.10 | Restricted visibility of two Privacy Groups (light blue and blue) | 13 |
| 2.11 | Besu and Tessera pair nodes administrator can give access to other Tenants, i.e., users. | 13 |
| 2.12 | EBSI interaction flow | 14 |
| 3.1 | Ideally, Monokee will have its own DID method (did:monokee), through which will be generated identifiers that will hide (at least, as far as the user is concerned) the blockchain where it is located. | 18 |
| 3.2 | Solution visual representation | 19 |
| 3.3 | Snippet of the callAPI function. | 23 |
| 3.4 | The functions used to generate the revocation tokens (in <code>utils.ts</code>). The <code>baseToken</code> is the private one, and is a concatenation of two random UUIDs. The <code>publicToken</code> is the derived by the <code>baseToken</code> in this manner: <code>base32(sha256(baseToken)).replaceAll("=", "")</code> | 25 |
| 3.5 | EBSI transaction timestamp example | 29 |
| 3.6 | All files' tests coverage percentages | 30 |
| 3.7 | Visual representation and statemets coverage | 30 |
| 3.8 | Branches, functions, and lines coverage details | 30 |
| 3.9 | The Holder page (Keys section) | 33 |
| 3.10 | The Issuer page (Issue section) | 34 |
| 3.11 | The Verifier page (Verifications section) | 35 |
| 3.12 | The Contracts page | 36 |
| 3.13 | The Diploma page | 36 |
| 3.14 | The EBSI page | 37 |

List of Tables

2.1 Analyzed kits 16

Chapter 1

Introduction

This chapter will introduce the problem: what we are analyzing, why this problem exists, how it is defined, and how it can be resolved. Also we present the company, the internship, and the work methodology.

1.1 The problem

As stated in the abstract, the main problem is preserving the ownership of people's data. In order to achieve this objective, we will pass through problems like interoperability, privacy safeguarding, law compliance, security, and others.

Assuming we can create a system where people hold their credentials (called Verifiable Credentials):

1. **Interoperability:** how can these credentials be shown to and verified in the same manner by different actors?
2. **Privacy:** can we demonstrate something without revealing it, preserving our privacy this way?
3. **Law compliance:** is it possible to save on blockchain people's data, or are we going against specific privacy laws?
4. **Security:** are credentials susceptible to attacks from hackers trying to steal our data?

Thanks to Self-Sovereign Identity, we can give a positive answer to all of these questions, but as is often the case, we have to deal with compromises.

Worldwide scenario. The current situation is clear. Every time we interact with a new website, we may want to interact with it, and to do so, we have to register to create a profile. In this phase, we have to give our data to the company, and they will be stored in their databases.

Problem identification. Let us now try to answer the previous questions to check how the present context is managed:

1. **Interoperability:** we could have two cases. In the first one, we use a technology that enables us to use our existing account on multiple websites, which integrates this solution, for example, "Sign-Up with Google". Here our data is owned by

Google, which shares them (if we grant permission) with third parties, and no one prohibits third parties from keeping our shared data saved. In the second case, we must register each time if the third party does not integrate other "Sign-Up with *" solutions. In both cases, third parties can collect our data (in the first case, Google explicitly knows our interests, but there is a minimum degree of interoperability). Also, in most cases, companies will let us create multiple accounts without verifying our data (one exception to this is the use of KYC).

2. **Privacy:** in some cases, we must show our data with complete transparency: for example, the police stop us on the street and ask for our details. Nevertheless, let us suppose we want to demonstrate something without revealing the details. For example, someone has graduated and wants to demonstrate it without revealing his final grade. We can do this thanks to a cryptography method called *Zero-Knowledge Proof*. However, this has not yet been implemented in most current systems.
3. **Law compliance:** if we consider saving users' data in blockchains, this problem does not exist as we examine centralized systems which do not use them. By the way, of course, there are privacy laws companies must follow (like GDPR).
4. **Security:** our information is stored in databases. With a data breach, considering a centralized system, a malicious actor can access all users' data at once. Sadly, this happens often. So often that someone has made a website where anyone can check if his data has been stolen online at least once ([Have I Been Pwned?](#)).

Problem statement. With the above considerations, it is clear that the existing systems work but could be significantly improved. In fact, interoperability enhancement would mean privacy and security penalization. Compromises exist, but if the system is well designed, they can be significantly reduced or at least moved to less dangerous areas. Here, the need for a more secure way to store user data arises. A way that intersects the analyzed points, bringing new power to people and reducing that of companies. This is the Self-Sovereign Identity's principle, which the developed solution will leverage.

Approaches. SSI concept is pretty simple, as opposed to its (in development) implementation. Everyone has different relationships or unique sets of identifying information. This information could include birth date, citizenship, university degrees, or business licenses. In the physical world, these are represented as cards and certificates that the identity holder holds in their wallet or a safe place like a safety deposit box. They are presented when the person needs to prove their identity or something about it. Self-sovereign identity (SSI) brings the same freedom and personal autonomy to the internet in a safe and trustworthy identity management system. SSI means the individual (or organization) manages the elements that make up their identity, and he digitally controls access to those credentials, called Verifiable Credentials (or VCs). They are digital representations of information that can be verified by a third party.

This is achievable by involving three participants:

1. **Holder:** the holder is an individual in the scenario, although it can also be an

organization/company. The holder is the entity that holds the credential.¹

2. **Issuer:** the issuer is the institution, be it a company, certifier body, or governmental organization, that has been awarded a level of trust to provide information (i.e., a public body that issued a passport)
3. **Verifier:** the verifier is the individual, organization, company, or government with whom the holder must prove information's legitimacy and trustworthiness.

The Verifiable Data Registry (VDR) grants the trust: here are stored schemas and identifiers (linked to the credentials) that the verifiers use to check data validity without the issuer's intervention.

To make a preliminary check of this solution's viability, let us try to answer the previous four questions, considering the new scenario:

1. **Interoperability:** with standards definition, credentials can be presented to verifiers by holders, in the same manner each time. Examples of standards could be credentials schemas (e.g., defining which fields are mandatory) and verification policies (i.e., how the credentials are verified).
2. **Privacy:** as already stated, we can demonstrate something without revealing its details with *Zero-Knowledge Proofs*. This technology has already found applications and implementations in blockchains (e.g., mixers, ZK-rollups, or ZK-games like Dark Forest), so a decentralized system that leverages ZKPs is buildable.
3. **Law compliance:** as will be read later in the paper, this is one of the most challenging points of the full SSI integration with blockchains because of its transparency nature. Everything is registered and immutable, so we must choose what to register and what not. Again, compromises are needed.
4. **Security:** as users hold credentials, as long as they are not saved in centralized servers, significant data breaches (targeting databases) would happen way less often. The user is responsible for his information security, and secure communication protocols will enhance it.

After quickly drafting these reflections, it can be said that SSI principles fit our problem requests, so a solution that aims to solve them can be tried to be developed. These analyzed points are well discussed in an article by Christopher Allen called "The path to Self-Sovereign Identity", where he defines the "Ten Principles of Self-Sovereign Identity".

1.2 Basic use cases

After addressing the problem and trying to provide answers to the initial questions, it is possible to start thinking about the first use cases. Obviously, the minimum requirement is credentials involvement: each time a user has to demonstrate some information, SSI could theoretically be leveraged.

In the first part of the internship, the focus has been (after the SSI primitives study) on use cases. They can be grouped into these two macro-categories:

¹Not always the holder and credential's subject coincide: for example, someone could hold a verifiable credential of his dog.

Academics. Here can be included all the uses about, for instance, university. A lot of them can be thought about and analyzed, but the most interesting which have been examined are these:

1. **Exams and Diplomas emission:** students can present their credentials (badge) to the university to register for exams. Each exam result would be another credential, and in order to access the diploma, he should present all the credentials related to passed exams (possibly wrapping them in a "presentation"). The final diploma would be another verifiable credential emitted by the university.
2. **Scholarships requests:** students can demonstrate they are eligible for facilitations by presenting their credentials to the university. This way, information pieces are easily checkable and verifiable, and procedures would be faster and less susceptible to errors. A "permit" credentials could be emitted, which would grant the student access to facilitations (e.g., canteen, money, discounts...)
3. **Discounts and university canteen:** it is evident that comfortable functions come into effect by processing the previous use case. Instead of creating accounts with university e-mail, students should present their credentials to services that offer facilitations, bringing interoperability and trust to each part.

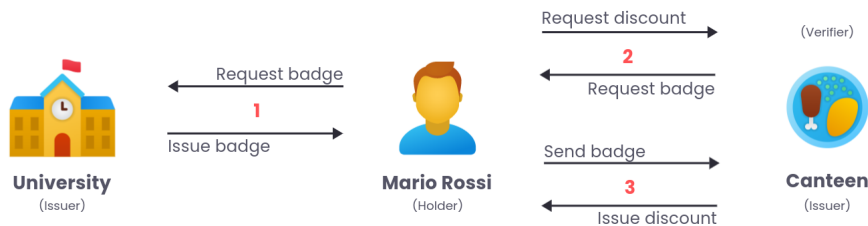


Figure 1.1: University canteen use case

Institutionals. Here can be placed all use cases involving the participation of national, European (or other unions), or global entities. Noteworthy examples could be:

1. **National ID:** this new system would replace physical identity cards with verifiable credentials. The municipality would issue them to the citizens, and the latter would use them to access all the national services or other services which request IDs.
2. **European SSI system:** this use case is currently in development and is called EBSI (European Blockchain Services Infrastructure). The final aim is to introduce the use of verifiable credentials in Europe and introduce new types of services to European citizens or improve the current ones.

These are just some examples of the possible use cases that can be developed, considering the model SSI offers. With the final product built during the internship (obviously, this would need additional integrations but gives a solid base), those listed are all viable scenarios.

1.3 Internship description

The previous sub-chapters clearly defined the problems and use-cases the product aims to involve and realize. This one will describe the structure of the internship, the company where it took place, and how the work was done.

1.3.1 The company

Officially, the company where the internship took place is called Athesys (from the Latin version of "Adige", i.e., "Athesis"), but actually, the job was about their startup, called **Monokee**.

Athesys

is a XaaS (Anything as a Service) integrator founded in 2010; they provide services such as database management, business intelligence, software development, security, and cloud.

In 2012 they began thinking about IAM (Identity and Access Management) solutions, delivering a product that, among many things, provides a Single Sign-On functionality across different domains.



Figure 1.2: Athesys logo

Monokee is born in 2017, and it is an innovative product-oriented startup that serves as an IAM for centralized and decentralized digital identities. In fact, its solution is hybrid: to the classic method (which involves using databases to store information), it intends to add SSI techniques, and this is where the internship comes in.



Figure 1.3: Monoquee logo

1.3.2 Internship objectives and planning

It is dutiful to specify that, at least initially, the objectives were not crystal clear. The overall concept of the internship itself was well defined, but the path to developing the whole solution was not.

The main objective was to develop **a software that enables the user to interact with verifiable credentials**. This type of software is named *Agent*, and users are intended as holders, issuers, and verifiers.

Before the internship, the Monoquee team searched for some existing solutions (unfortunately, not numerous) and found an SSI Kit developed by walt.id, a European company focused on SSI.

So, in the beginning, the steps to follow were:

1. Deep dive into SSI technology and its primitives;

2. Analysis of the problem and understanding of what is needed and how to use it for the final product;
3. Study of SSI Kit, provided by walt.id;
4. If it fits the needs, leverage it to develop the Agent (if not, develop a similar software);
5. If possible, integration into a web application Proof of Concept with blockchain's smart contracts.

The path to pursue became more apparent during the first two steps (technology study and problem analysis), so the final internship structure became this:

1. **Requirements analysis:** in this phase, SSI has been well studied and comprehended to understand the next steps. It has been divided into:
 - (a) **Technologies study:** understanding of the existing standards;
 - (b) **Solution conception:** definition of the following steps;
2. **SDK Development:** development of the library that serves as an abstraction of the existent SSI Kit, allows it to be used on a web application. Divided into:
 - (a) **Software development:** code development of main entities;
 - (b) **SSI Kit source code study:** needed mostly because of documentation lack;
 - (c) **Testing:** unit testing of the library main components;
3. **PoC Development:** final part of the internship, where has been developed a web application that merges the SSI Kit SDK with smart contracts with SSI features. Separated into:
 - (a) **Software development:** development of the web application (back-end and front-end);
 - (b) **SDK improvement for integration:** improvement of the SDK to better fit the web application needs;
 - (c) **Debug/UI-UX improvement:** final arrangements of the proof of concept.

In the following Gantt chart are outlined the timings of each step:

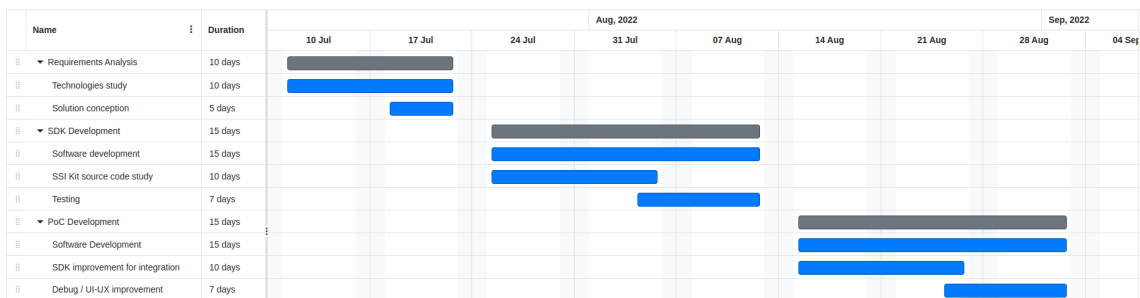


Figure 1.4: Internship structure

Chapter 2

State of the art and technology background

This chapter presents the pre-concepts needed to comprehend this paper's content fully. As is understandable from the introduction, they are about Self-Sovereign Identity and blockchains. In addition, state of the art will be analyzed to see what has already been done and what can be improved.

2.1 Technology concepts

This section will explain in detail SSI and blockchain technologies.

2.1.1 Self-Sovereign Identity concepts

Here can be read a brief reprise of what has already been saying about Self-Sovereign Identity and a description of its main primitives: VCs, VPs, and DIDs.

Self-Sovereign Identity

Self-Sovereign Identity is an approach to digital identity that gives individuals control over their data. SSI addresses the difficulty of establishing trust in interaction and allows people to interact in the digital world with the same freedom and ability to trust as they have in the offline world.

To be trusted, a party in an interaction will present credentials to other parties, and those parties can verify that the credentials come from an **issuer** they trust. This way, the **verifier**'s trust in the issuer is transferred to the credential **holder** (or **prover**). This basic structure of SSI with three participants is sometimes called the "triangle of trust.", simply because you need an element of trust among these entities for them to work together.

While this does not mean that there is a legal partnership or understanding between the entities involved, it does mean that each of the entities is willing to examine the credibility of the other, and this implicit trust is what constitutes this term.

The Verifiable Credentials Flow

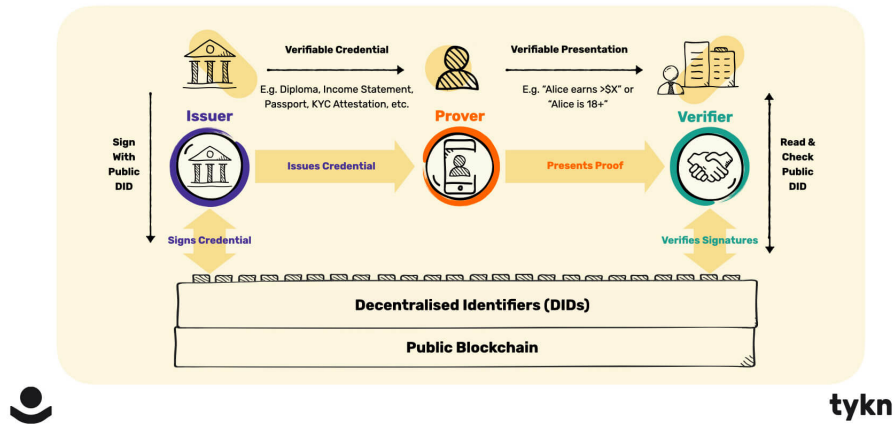


Figure 2.1: The triangle of trust: Prover, Issuer, and Verifier (by Tykn)

Verifiable Credential (VC)

A verifiable credential can represent all of the same information that a physical credential represents. The addition of technologies, such as digital signatures, makes verifiable credentials more tamper-evident and more trustworthy than their physical counterparts.

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "http://example.edu/credentials/1872",
  "type": ["VerifiableCredential", "AlumniCredential"],
  "issuer": "https://example.edu/issuers/565049",
  "issuanceDate": "2010-01-01T19:23:24Z",
  "credentialSubject": {
    "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
    "alumniOf": {
      "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
      "name": [{
        "value": "Example University",
        "lang": "en"
      }, {
        "value": "Exemple d'Université",
        "lang": "fr"
      }]
    }
  },
  "proof": {
    "type": "RsaSignature2018",
    "created": "2017-06-18T21:19:10Z",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "https://example.edu/issuers/565049#key-1",
    "jws": "eyJhbGciOiJIUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsyYjY0IiwuTCYt5XsITJX1CxpCT8yAV-TVkIEq_PbCh0MgsLfRoPsnsgw5WEuts0lmq-pQy7UJiN5mgRxD-wUcX16dUEMGlV50aqzpqh4Qktb3rk-BuQy72IFL0qV0G_zS245-kronKb78cPN25DGLcTwLtjPAYuNzVBAh4vGHSrQyHUdBBPM"
  }
}
```

Figure 2.2: Example of verifiable credential (VC)

Holders of verifiable credentials can generate verifiable presentations and then share these verifiable presentations with verifiers to prove they possess verifiable credentials with certain characteristics.

Both verifiable credentials and verifiable presentations can be transmitted rapidly, making them more convenient than their physical counterparts when trying to establish trust at a distance. The three main components of a VC are:

1. **Metadata:** cryptographically signed by the issuer. It describes the credential properties, such as the issuer, the subject, the expiry date and time, a public key to use for verification purposes, the revocation mechanism, and other information;
2. **Claims:** a statement made about a subject. Example: “Janice’s date of birth is 01/01/1990.”
3. **Proofs:** a proof is data about the identity holder that allows others to verify the source of the data (i.e., the issuer), check that the data belongs to (only) the holder, that the data has not been tampered with, and finally, that the issuer has not revoked the data.

Verifiable Presentation (VP)

A verifiable presentation expresses data from one or more verifiable credentials and is packaged in such a way that the authorship of the data is verifiable. If verifiable credentials are presented directly, they become verifiable presentations. Data formats derived from verifiable credentials that are cryptographically verifiable but do not themselves contain verifiable credentials might also be verifiable presentations.

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "type": "VerifiablePresentation",
  "verifiableCredential": [{
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "id": "http://example.edu/credentials/1872",
    "type": ["VerifiableCredential", "AlumniCredential"],
    "issuer": "https://example.edu/issuers/565049",
    "issuanceDate": "2010-01-01T19:23:24Z",
    "credentialSubject": {
      "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
      "alumniOf": {
        "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
        "name": [{
          "value": "Example University",
          "lang": "en"
        }, {
          "value": "Exemple d'Université",
          "lang": "fr"
        }]
      }
    }
  }],
  "proof": {
    "verifiableCredentialProof..."
  }
},
{
  "proof": {
    "verifiablePresentationProof..."
  }
}
```

Figure 2.3: Example of verifiable presentation (VP)

The data in a presentation is often about the same subject but might have been issued by multiple issuers. The aggregation of this information typically expresses an aspect of a person, organization, or entity.

Decentralized Identifier (DID)

Decentralized identifiers are a new type of identifier that guarantees a verifiable, decentralized digital identity. A DID refers to any subject (e.g., a person, an organization, a data model, an abstract entity...).

DIDs are decoupled from centralized registries, identity providers, and certification authorities. Specifically, while other parties can be used to retrieve information about a DID, the design allows the controller of a DID to demonstrate control over it without requiring permission from other parties.



Figure 2.4: Example of a DID

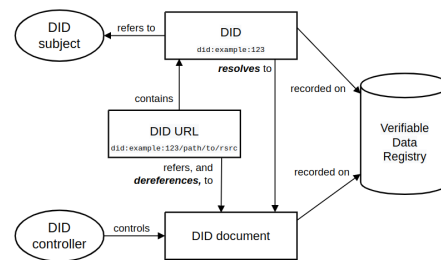


Figure 2.5: DID architecture overview and basic components relationship

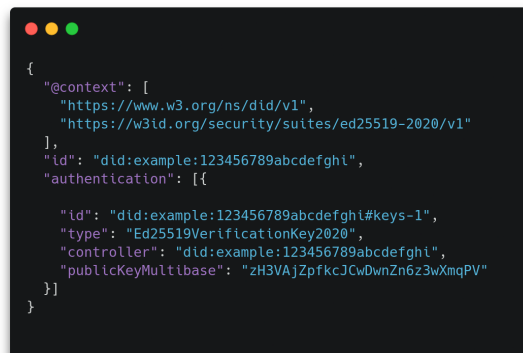


Figure 2.6: Example of DID document

DIDs are Uniform Resource Identifiers (URIs) that associate a DID subject with a DID document that enables trusted interactions associated with that subject. Each DID document may contain encrypted material, verification methods, or services, which provide a set of mechanisms that allow a DID controller to demonstrate control of the DID. Services enable trusted interactions associated with the subject of the DID. A DID may provide the means to return the DID subject itself if the DID subject is an information resource such as a data model.

A DID is a simple text string consisting of three parts: 1) the DID URI scheme

identifier ¹, 2) the identifier for the DID method ², and 3) the DID method-specific identifier.

JavaScript Object Notation (JSON)

The VC, VP and DID document code examples showed above are all in JSON. The JSON is a lightweight data-interchange format. It is easy for humans to read and write, and for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is built on two structures:

- * A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- * An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

2.1.2 Blockchain concepts

Here we will introduce the main concepts of blockchain technology, essentials to understand the PoC development phase and some SDK features.

Blockchain

The blockchain is a shared, immutable database structured in the form of a chain of blocks, each of which contains a set of information. In essence, blockchains represent digital ledgers and perform different functions.

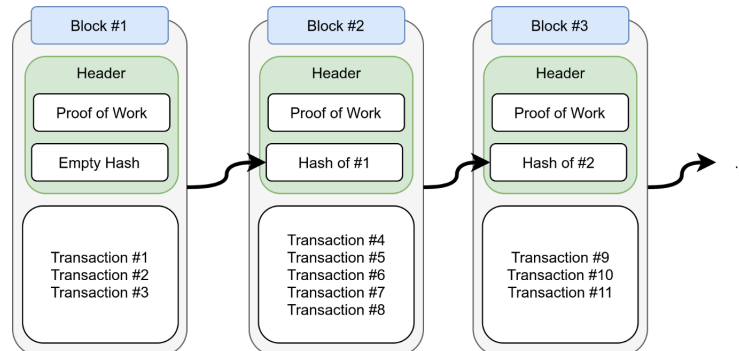


Figure 2.7: Simple blockchain visualization

Physically it is composed of multiple **nodes**, i.e., computers that run the software (Client) of the blockchain. When a user wants to interact with the blockchain, he sends a **transaction** to one of the nodes. This transaction will then reach a pool

¹The formal syntax of a decentralized identifier. The generic DID scheme begins with the prefix *did:*.

²A definition of how a specific DID method scheme is implemented.

with all the transactions in the pending state, and the nodes that are dealing with the consensus part will take care of updating the state of the blockchain by inserting several transactions (meeting certain conditions) within the new **block** that will be added to the chain.

Its main features are data digitalization, decentralization, disintermediation, transfer traceability and programmability, transparency/verifiability, and immutability.

Permissionless and permissioned blockchains

We can divide blockchains into two main categories: permissionless and permissioned. This division is based on the access to the network.

- * **Permissionless blockchains** are the most popular model. As its name implies, these networks allow access to anyone and are decentralized and public. Consequently, everyone can run a node or connect to the blockchain.

This accessibility implies a trade-off on speed; these networks are often slower than their permissioned counterparts, with fewer members, and transactions are validated by everyone running a connected node. The primary consensus mechanisms are Proof-of-Work (PoW) and Proof-of-Stake (PoS) ³.

What is of most interest in this scenario is that in a permissionless blockchain, everyone can interact, and data is public, so **preserving privacy becomes difficult**.

- * **Permissioned blockchains** are private networks that require permission to join. They are usually run by a single organization or a consortium of organizations. The main advantage of permissioned blockchains is speed. They are faster than permissionless blockchains because they have fewer members and transactions are validated by a smaller number of nodes. The main consensus mechanisms are Raft and Practical Byzantine Fault Tolerance (PBFT).

The main peculiarity of permissioned blockchains is that they are not decentralized and are not public. This means that only a limited number of people can interact with the blockchain, and data is not public, so **privacy can be preserved**.

Ethereum

Ethereum is a permissionless blockchain, i.e., open to anyone who wants to interact with it: the trade-off is the introduction of fees, to be paid every time anyone wants to change the state of the Ethereum Virtual Machine (EVM), to mitigate the problems that the permissionless factor introduces (e.g., transaction spam). In the read-only case, no fee needs to be paid. Anyone can then view what is happening in the blockchain and, more importantly, use it. To do so, a user has to generate a **wallet** (i.e., an address in the blockchain), transfer some funds to it from outside ⁴ so that fees can be paid, and start interacting with the decentralized applications that are already developed or transfer funds to other wallets. In fact, one of the main features of Ethereum is **smart contracts**: they are programs that run on the Ethereum blockchain. They are a collection of code (functions) and data (state) that resides at a specific address on the Ethereum blockchain.

³PoW involves hashing (mining) power, PoS voting (using blockchain coins) power, through validator nodes.

⁴Usually from a centralized exchange, where cryptocurrencies can be traded for fiat currencies like EUR or USD, or from another blockchain.

Hyperledger

Hyperledger Foundation is a nonprofit organization that combines all the resources and infrastructure needed to ensure thriving and stable ecosystems around open-source software blockchain projects.

Hyperledger Foundation staff are part of the larger **Linux Foundation** team with years of experience providing management services for programs for open-source projects.

Hyperledger Besu

Hyperledger Besu is an Ethereum client designed to be enterprise-friendly for use cases of public and private permissioned networks, which require secure transaction processing and high performance.

The Besu blockchain, therefore, is EVM compatible: from the perspective of developers and users, interaction with it will be very similar to interaction with Ethereum. It places particular emphasis, however, on **privacy and permissioning** features; in fact, only those who are authorized (i.e., those who own a connected node) can interact with the system, which is the primary difference from Ethereum.

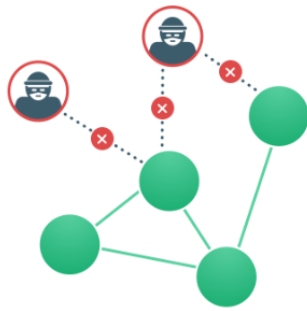


Figure 2.8: Only allowed users can participate in the network

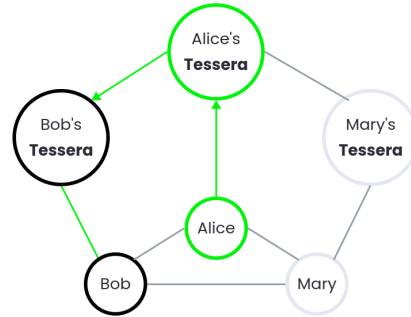


Figure 2.9: Mary cannot see the private transaction sent from Alice to Bob

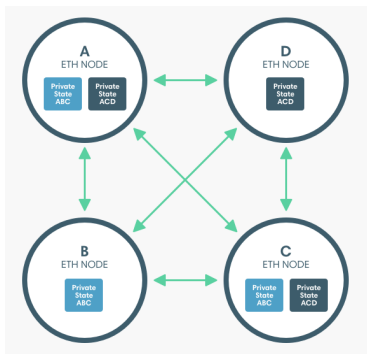


Figure 2.10: Restricted visibility of two Privacy Groups (light blue and blue)

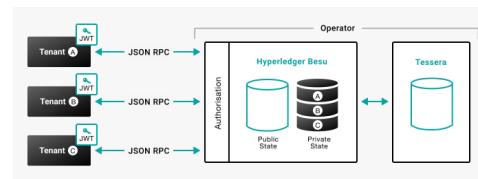


Figure 2.11: Besu and Tessera pair nodes administrator can give access to other Tenants, i.e., users.

Privacy is enabled both externally to the network and internally: through private transactions, not all nodes can access certain information, and nodes that want to take advantage of private transactions must have an associated **Tessera** node, which will take care of the cryptographic part.

Even **Privacy Groups** can be created: those who do not belong to the group cannot access particular data. In addition, another interesting feature is that of Multi-Tenant management: multiple participants can use the same Besu and Tessera node through a dedicated user system.

Hyperledger Fabric

Hyperledger Fabric is an enterprise-grade, proven, and open distributed ledger platform (DLT). It provides advanced privacy controls so that only the shareable data is transmitted among network participants, known as "authorized".

It offers a modular architecture that makes available components (mechanism for consensus, services for joining and managing blockchain members) that can be activated within a blockchain with plug-and-play logic. It can be said to be very similar to Besu (also a permissioned and privacy-oriented), with the big difference being that it is not EVM compatible so the smart contracts will be written in languages such as Java and Go instead of Solidity.

2.1.3 Libraries and Stack involved

After analyzing all the necessary pre-concepts, we can now introduce the last ones: EBSI, which can be used as VDR, and walt.id, the software suite that includes the SSI Kit.

EBSI

European Blockchain Services Infrastructure (EBSI) is a service infrastructure for managing European citizens' identity and education credentials. These use cases aim to facilitate the mobility of students, young professionals, and entrepreneurs, as well as ensure and verify the authenticity of digital information in different sectors.

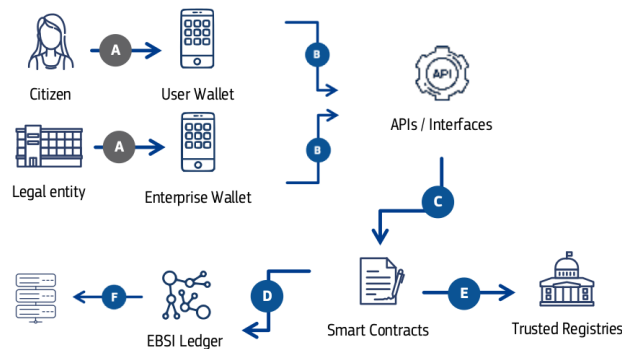


Figure 2.12: EBSI interaction flow

Its structure is rather complex, consisting of two essential layers: the bottom layer is made up of EBSI's blockchain, which holds all the credentials and smart contracts with which one can interact with them, while the upper layer by a microservices architecture

and Application Programming Interface (API), which interface with the blockchain. In practice, to talk to the EBSI blockchain (if you do not have a node), you must interface through the microservices architecture and API. The blockchains used by EBSI are Hyperledger Fabric and Hyperledger Besu.

Finally, in the blockchain are already deployed a series of smart contracts, serving as a registry for DIDs (and DID documents) and for trusted applications, issuers, ledgers, and policies.

walt.id SSI Kit

The walt.id software suite, an open-source project developed by a European team, consists of a set of kits that allows us to add SSI functionalities to our product. Specifically, by leveraging the SSI Kit, it will be possible to generate encrypted keys, register and resolve DIDs, create, issue, submit and verify VCs, and much more. This kit can be used via Command Line Interface (CLI) or REST API. Currently, the kit supports `did:key`, `did:web`, and `did:ebsi` methods (the latter allows us to talk to the EBSI blockchain). Since, by default, it dialogues with other APIs (e.g., EBSI's API), if we wanted to register, for example, a DID in a mainnet that has not yet been registered, such as Sovrin, we would need to add the `did:sov` method to the list of supported methods, also implementing the dialogue with Sovrin's API. Doing this would require a new module development, integrating it with the SSI Kit, and developing something similar to what has been done for EBSI, although the complexity is unknown.

2.2 State of the art

Taking back the purpose of this paper, we want to build a hybrid solution, merging SSI with blockchain smart contracts. Currently, within SSI, most blockchains are used only as VDR (so to store DIDs, VC verification policies, schemas...).

The final software (PoC) has to leverage a kit to offer SSI functionalities (SDK) and combine these with blockchain features to automate verifications and possibly facilitate certain issuing cases (smart contract suite).

That said, we can now illustrate what already exists and what has to be crafted.

2.2.1 Complete solution

"Complete solution" means a software that already merges an SSI SDK with a smart contract suite. Unfortunately, during the first two weeks of research and situation study, we did not find anything already built, probably because of the technologies involved novelty.

As a result, we must find existent SDKs and smart contract standards to build the desired solution and, if not, build everything from scratch.

2.2.2 SSI Kits

As already stated and introduced, SSI Kit by walt.id is a Swiss Army knife for SSI primitives. However, it has been chosen after analyzing other existing solutions.

Initially, we were looking for a kit that provides multichain support (i.e., the possibility to interact with more DIDs across multiple blockchains). Unfortunately, an equivalent of the SSI Kit with multiple chains support does not exist, and every other found

kit supports at most one blockchain. The Decentralized Identity Foundation (DIF) develops the most interoperable tools we found in the SSI scope: Universal Resolver and Universal Registrar.

We now describe all the solutions we found and specify why we choose the SSI Kit by walt.id.

Firstly, all the existing kits support the two main DID methods:

- * **did:key**: a non-registry based method that generates a DID from a cryptographic key. It is the simplest method;
- * **did:web**: a method that generates a DID that enables the use of an existing web domain as identifier.

Other than SSI Kit by walt.id, we found two SDKs developed by MATTR and Veramo Labs. In this table, we will outline the kit, which DID method is supported other than the two discussed, the documentation status, and whether the kit is free to use.

| SSI Kit | DID method | Documentation | Free to use | Support |
|---------|-----------------------|-------------------------|-------------|-------------|
| walt.id | did:ebsi | Improvable ⁵ | Yes | Team, Slack |
| MATTR | did:ion ⁶ | Good | No | E-mail |
| Veramo | did:ethr ⁷ | Improvable | Yes | Discord |

Table 2.1: Analyzed kits

To summarize:

- * **MATTR** SDK adds support for **did:ion** method. Its biggest flaw is the lack of support from the team: the only method to reach them is by sending an e-mail. In addition, the kit is not free to use.
- * **Veramo** SDK supports the **did:eth** method. We need to join the Discord server (not so active) to ask for support, the SDK is free, but the documentation could be improved.
- * **walt.id** SSI Kit adds support for the EBSI blockchain. It is free to use, and the documentation is improvable, but as the kit is in development, the team is very active in the Slack channel (also, Monokee had direct contact with the walt.id team).

After the research, we choose walt.id SSI Kit, mainly because the kit offered more functionalities and was more customizable. Additionally, it supported EBSI, which was a good "nice-to-have" for Monokee, as in contact with the University of Naples Federico II to build a project interacting with the European blockchain.

Finally, the team was very active and chose to support us in developing the SDK for walt.id SSI Kit.

⁵Improvable: not everything is documented so we must read the source code or contact the team

⁶DID method to interact with ION, a Bitcoin sidechain

⁷DID method to interact with Ethereum

2.2.3 Smart contract suites

Regarding the use of smart contracts in the SSI environment, we first need to know how they can be used.

The roles they can assume, following the SSI model, are two: issuer and verifier (holder would not make much sense). The role we already know they can assume is a register of DID and events like verification or revocation.

Let us now try to think about what should a smart contract need in order to be an issuer or a verifier:

- * **Issuer:** he has to be able to produce a signature that confirms who is issuing owns the DID (to certify his identity), other than to generate a VC. This has to be made with the private key of that DID, and everyone can verify the signer's identity with the public key. After a discussion with the Monokee team, we decided to avoid implementing this unless we find something already done.
- * **Verifier:** he has to be able to verify a VC, which means he has to take it in input and verify it using the verification method described in the appropriate field. Also, to enhance trust, it should sign the verification result with the private key of its DID. Again, these functions could be hard to implement inside a smart contract, so we came to the same conclusion.

Our research uncovered two libraries: one for the leading and most used patterns and one for the SSI features.

The first one is an extensive collection of patterns and security standards developed by **OpenZeppelin**. It ranges from the simplest things, such as a counter standard contract, to the most complex, such as Ethereum Request for Comments (ERC) implementations (ERC-20 is one of the most famous, and it is the standard for Ethereum fungible tokens) For example, we will use the ERC-721 implementation (Non-Fungible Tokens, or NFTs) to implement one of the use cases.

The second one is a suite of smart contracts that aims to bring SSI on-chain. It is called **Verite** and is being developed by Centre (to give a background, Centre is the company that developed USDC, one of the leading stablecoins in the cryptocurrency market). As it stands, the library only contains a smart contract that serves as a verification results registry, and we decided to take it and adapt it to our purposes.

Since, during our research, we did not find an implementation of on-chain issuer or verifier agents, we decided to re-use already existing smart contract libraries and build some simple auxiliary smart contracts we needed in order to implement on-chain registries to at least make some actions more immediate (like verifications and revocations).

Chapter 3

Solution

Now we discuss the path we decided to take, how we developed the software, and the technologies we leveraged. Then, we outline the final achievements and what can be done to enhance the PoC potential.

3.1 Solution proposal

After the conducted analysis in the first two weeks, we concluded that building a new system from zero would have needed too much time and effort, and especially would have required specific advanced skills we did not have.

First, we recall that we are building agents for verifiable credentials interaction. In a full stack product, our solution is placed between the users, who use secure communication protocols¹, and VDRs², which store the DIDs, credentials schema, verification policies, and more.

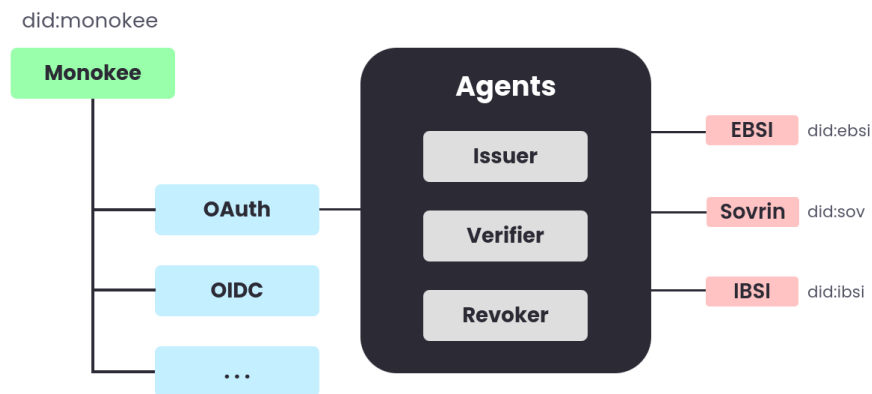


Figure 3.1: Ideally, Monokee will have its own DID method (did:monokee), through which will be generated identifiers that will hide (at least, as far as the user is concerned) the blockchain where it is located.

¹For example, OAuth or OIDC, as can be seen in the Figure 3.1

²For example EBSI, Sovrin or IBSI, blockchain used for SSI purposes

The final software structure has three main components:

- * **Frontend**: it allows the user to interact with the system's core functionalities and serves as an interface for every SSI Kit SDK function.
- * **Backend**: it is needed for security, as we will analyze further, and for cryptographic functions that the frontend could not execute.
- * **SSI Kit SDK**: it exposes all the SSI functionalities, and enables the user to create keys and DIDs, issue VC, present them as VPs, and more.
- * **Smart Contracts**: for what concerns SSI Kit integration, the contracts serve as trusted verifiers and verification results register. Some contracts emit ERC-721 tokens, which let the user request the diploma, but they will not be discussed here.

In the **figure** can be seen the visualization of the system final architecture.

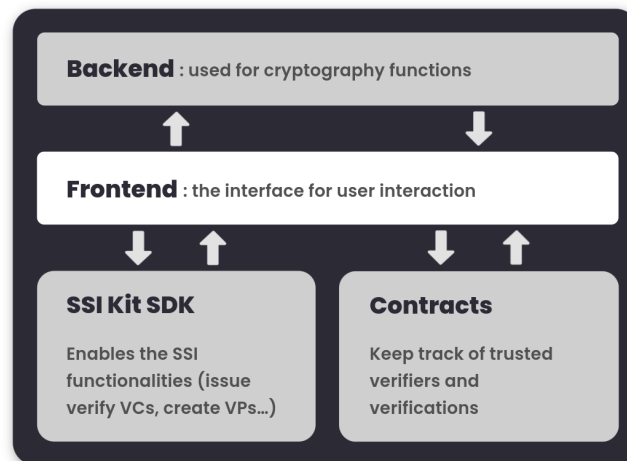


Figure 3.2: Solution visual representation

3.2 Solution development

In the following sections, we discuss the technologies we used to build the solution, and we explain the main functionalities of the system.

3.2.1 Technologies and Tools

Before explaining the solution, we list the languages and tools we leveraged to develop it, for both SSI Kit SDK and PoC.

Common tools and languages

- * **Typescript**: a strongly typed programming language that builds on JavaScript. It was chosen because the other Monokee's modules were in Typescript, so it would have been easier for the team to integrate. Also, it is very convenient for its strongly typed nature;
- * **JSON**: as already covered in [Chapter 2](#), JSON is a lightweight data-interchange format. It has been extensively used, mainly for credentials representation and API calls;
- * **Node.js**: a JavaScript runtime built on Chrome's V8 JavaScript engine. It has been used for code execution;
- * **npm**: a package manager for the JavaScript programming language. It has been used to manage the dependencies of the projects;
- * **Git**: a free and open-source distributed version control system used for tracking and collaboration purposes;
- * **Visual Studio Code**: the Integrated Development Environment (IDE) we have used for the solution development.

SSI Kit SDK³

- * **jest**: a JavaScript testing framework. It has been used to test the SSI Kit SDK components;
- * **waltid-ssikit**: the library written in Kotlin/Java that provides the SSI functionalities set. The developed SDK is a Typescript wrapper of this library;
- * **axios**: a promise-based HTTP client for the browser and node.js, used to make API calls to waltid-ssikit;
- * **uuid**: a library used to generate RFC-compliant Universally Unique Identifiers (UUIDs);
- * **rfc4648**: a library used to encode and decode data in Base32 format;
- * **sha256**: a library used to generate SHA-256 hashes;
- * **nacl**: a library used to decode UTF8 strings;

³`uuid`, `rfc4648`, `sha256`, and `nacl` have been used just to generate tokens used for credentials revocation, as can be read [here](#)

Frontend

- * **React.js**: a JavaScript library for building user interfaces. It has been used to build the frontend;
- * **Chakra-UI**: a simple, modular and accessible components library, used with **React.js** to build the frontend.
- * **ethers**: a library used to interact with Ethereum Virtual Machine compatible blockchains;
- * **wagmi**: a collection of React Hooks containing everything needed to start working with Ethereum; it has been used to interact with the smart contracts;
- * **RainbowKit**: RainbowKit is a React library that makes it easy to add the wallet connection, e.g., for Metamask integration.
- * **GraphQL**: the query language used by The Graph;
- * **ssikit-sdk**: the developed Typescript SDK used to interact with the SSI Kit library.
- * **The Graph**: a decentralized protocol for indexing and querying data from blockchains, starting with Ethereum. It makes it possible to query data that is difficult to query directly. It has been used to query The deployed smart contracts.
- * **smart contracts suite**: a collection of smart contracts used to register the verifications and the verification results on-chain.

Backend

- * **Express.js**: a web application framework for Node.js. It has been used to build the backend, where the cryptographic functions are executed; the frontend calls them through API calls;
- * **ssikit-sdk**: the developed Typescript SDK used to interact with the SSI Kit library.
- * **jose**: a library used to encode and decode JSON Web Tokens (JWTs), which have been used to represent private and public keys;
- * **nodemon**: a tool that automatically restarts the node application when file changes in the directory are detected.

3.2.2 SSI Kit SDK development

In this section we describe the SSI Kit SDK development, which is the core of the solution. The SDK is a Typescript wrapper of the `waltid-ssikit` library, which is written in Kotlin/Java. The SDK exposes all the functionalities of the library, and it is used by the frontend and the backend to interact with the SSI Kit.

walt.id SSI Kit

First, we need to know how to interact with it to understand how to build the SDK. The kit gives us three options:

- * **CLI Tool**: it offers a rich set of commands to run the entire functionality the SSI Kit provides. The CLI tool can be used by running the Docker container or the executable by the local build.

- * **Dependency (JVM)**: it can be used directly as JVM-dependency via Maven or Gradle.
- * **REST API**: it can be run as a service, so an application can access its functionalities via REST API.

As we decided to build a Typescript SDK, the REST API is the most convenient way to access the kit's functionalities: if we chose CLI, every time we call an SDK method, we should have fired a shell command translating the method execution, and it would be the most uncomfortable option. The Dependency option, in addition, is immediately discardable as we do not use Maven or Gradle (the application is not Java/Kotlin based).

The REST API service, instead, is very immediate to integrate into an SDK. In fact, we previously defined the SDK as a "wrapper" because the SDK methods (at least, the great majority) forward their execution as an API call to the underlying kit. This makes the interaction with the kit a lot easier for Javascript/Typescript application based.

Let us now describe how the SSI Kit API is structured to explain later the choices made for the SDK development.

Structure. The API is divided in five main components:

- * **Signatory**: the component that manages the issuing and revocation of verifiable credentials. Also, it can list the credential templates used to issue them;
- * **Custodian**: gives a CRUD interface for keys, DIDs and credentials. Here also can be generated VPs;
- * **Auditor**: enables the user to manage verification policies, and verify any verifiable credential or presentation;
- * **ESSIF**: enables the user to interact with EBSI blockchain (e.g., register a DID);
- * **Bonus: Core API**: It is a set of functionalities chosen from the above components. However, the team writes in the documentation that it will not be maintained. For this reason we decided to not build a class for this component.

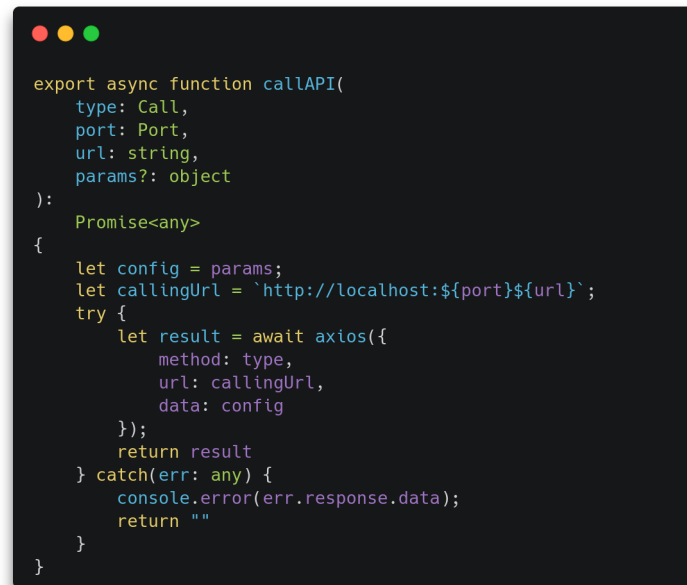
Assuming this, we can say it could be a good idea to build a class for each component to organize the SDK better and separate the code.

SSI Kit Typescript SDK

After analyzing the SSI Kit structure, we can describe how we developed the SDK. As stated above, we decided to build a class for each SSI Kit component to modularize the SDK. In every class, we put a method corresponding to an API call.

For API calls, we use Axios, through which we perform a call in each SDK method. So a good thing to do is to create a generalized function, called in every method, without writing the same code where possible.

In the [figure](#) can be seen a snippet of the `callAPI` function, which is used to perform the API calls.



```
export async function callAPI(  
  type: Call,  
  port: Port,  
  url: string,  
  params?: object  
):  
  Promise<any>  
{  
  let config = params;  
  let callingUrl = `http://localhost:${port}${url}`;  
  try {  
    let result = await axios({  
      method: type,  
      url: callingUrl,  
      data: config  
    });  
    return result  
  } catch(err: any) {  
    console.error(err.response.data);  
    return ""  
  }  
}
```

Figure 3.3: Snippet of the callAPI function.

As input, it takes four parameters:

- * **type**: the HTTP method to use; it can be "GET", "POST", "PUT" or "DELETE";
- * **port**: the SSI Kit service needs to be run to use the SDK. It then makes available four ports, which are used to access the different components of the SSI Kit. The **port** parameter is used to specify which port to use; it can be
 - 7001 (Signatory);
 - 7002 (Custodian);
 - 7003 (Auditor);
 - 7004 (ESSIF);
 - 8080 (Universal Resolver);
- * **url**: the URL of the API call (e.g., "/v1/verify" to call the credentials verification method of the SSI Kit);
- * **params**: this is an optional parameter, which is used to pass additional info to the call if needed.

The return type is generic, as there are many types of results (keys, DIDs, VCs, booleans, and more). Also, it can be seen that the two parameters **type** and **port** have user defined types. These types and the **apiCall** function can be found inside the file **utils.ts**, which is a collection of types, functions and interfaces used by the SDK, as we analyze in the next section.

For the most input/output object parameters of the API calls, we created interfaces to

type them. This is useful to have a better understanding of the parameters, to avoid errors and to facilitate the use of the SDK.

For verifiable credentials, we did not, as their structure may change from instance to instance. We simply treat them as JSON objects.

We could leave everything as a JSON object, but we decided to create interfaces for the most important objects. Leaving them as JSON objects would have been a bad idea, as it would have been difficult to understand what the parameters are and what they do. The only pro would have been to avoid the creation of interfaces, but we think that the cons are more important.

Structure. The SDK is composed of five main classes. Four are the classes corresponding to the four previously described components of the SSI Kit, and the fifth is the one dedicated to the Universal Resolver. Other than this, the SDK offers a `utils.ts` file.

The SSI Kit SDK splits into three main folders:

- * **core:** it contains the main classes of the SDK, which are the ones that interact with the SSI Kit API; it also contains the `utils.ts` file, which is a collection of types, functions, and interfaces used by the SDK, and the `lib.ts` file, which contains two functions that perform multiple SDK calls (e.g., `registerDIDOnEBSI` which performs the entire onboarding procedure of a DID in the EBSI blockchain);
- * **interfaces:** it contains the interfaces used by the four SSI Kit class implementations;
- * **tests:** it contains the tests of the SDK.

Everything (function, type, interface) with the `export` declaration gets exported by the `index.ts` file, which is the entry point of the SDK.

Functionalities. After examining the SDK structure, the choices we made, and some technical detail, we can now fully explain its functionalities.

- * **Signatory:** this class serves as the issuer component in our SSI model. It implements five methods:

`issueCredential`: issues a verifiable credential;

- * **Params:** `IssueCredentialRequest`, which is an object where it is specified the template ID (chosen by those provided by the SSI Kit), the credential data and the proof configuration;
- * **Returns:** a W3C Verifiable Credential in JSON format;

`getVCTemplateIDs`: returns the IDs of all the SDK templates;

- * **Returns:** an array of strings, where each string is a template ID (e.g., "UniversityDegree"); these templates are usable in the `issueCredential` method;

`getVCTemplate`: it returns a specific template;

- * **Params:** the `templateId` string;
- * **Returns:** a verifiable credential template in JSON format;

isRevoked: checks if a verifiable credential is revoked;

- * **Params**: the corresponding `publicRevocationToken`;
- * **Returns**: the result of the check, which is an object with the fields `revoked` (boolean) and `token` (string);
- * **Notes**: the `publicRevocationToken` is findable inside the `credentialStatus` field of the VC. When an issuer issues a VC, the SDK generates a `private` and a `public` revocation token. The `private` has to be saved by the issuer, and can be used to revoke the credential. The `public` one is included in the VC, and everyone can use it to check if the VC has been revoked or not.



```
export function getRandomUUID(): string {
  return uuidv4();
}

export function createBaseToken(): string {
  return getRandomUUID() + getRandomUUID();
}

export function deriveRevocationToken(baseToken: string): string {
  return base32.stringify(
    sha256(nacl.decodeUTF8(baseToken))
  ).replaceAll("=", "");
}
```

Figure 3.4: The functions used to generate the revocation tokens (in `utils.ts`). The `baseToken` is the private one, and is a concatenation of two random UUIDs. The `publicToken` is the derived by the `baseToken` in this manner: `base32(sha256(baseToken)).replaceAll("=", "")`

revokeCredential: revokes a verifiable credential;

- * **Params**: the `privateRevocationToken`, owned only by the issuer of that VC;
- * **Returns**: the revocation result, which is a `boolean` (if `false`, something went wrong during the process).

- * **Custodian**: this class serves as the holder component in our SSI model. It implements eighteen methods, and manages keys, DIDs and credentials:

getKeys: gets the keys in the holders's wallet;

- * **Returns**: an `array` of `Key` objects, which are the keys owned by the holder;

getKey: gets a specific key in the holder's wallet;

- * **Params**: the `keyId` string, which is the key identifier;
- * **Returns**: the corresponding `Key` object;

generateKey : generates a new key;

- * **Params**: the **keyAlgorithm** string, that specifies the algorithm used to generate the key; the supported algorithms are **RSA**, **EdDSA_Ed25519** and **EdDSA_Secp256k1**;
- * **Returns**: the generated **Key** object;

deleteKey : deletes a key from the holder's wallet;

- * **Params**: the **key** parameter, which can be the **keyId** string or the **Key** object;
- * **Returns**: the result of the deletion, which is a **boolean**;

exportKey : exports a key (private or public) in the desired format;

- * **Params**: the **key** parameter (**id** string or **Key** object), the **format**, which can be **JWK** or **PEM**, and **exportPrivate** (if **true**, the private key is exported, otherwise the public one);
- * **Returns**: the exported key in **JSON** format;

importKey : imports a key in the holder's wallet;

- * **Params**: the **formattedKey** parameter (**JWK** or **PEM** format),
- * **Returns**: the **keyId** of the imported key;

getDIDs : gets the DIDs owned by the holder;

- * **Returns**: an array of **DID** strings;

getDID : gets a specific DID owned by the holder, from local storage;

- * **Params**: the **did** string;
- * **Returns**: the corresponding **DID JSON** with related metadata (e.g., **verificationMethod**, used to verify the DID signature);

createDID : creates a new DID;

- * **Params**: the **method** (**key**, **web** or **ebsi**), the **key** (object or id string) used to generate the DID, and two optionals (for **web** method): **didWebDomain** and **didWebPath**;
- * **Returns**: the created **DID** string;
- * **Notes**: the DID is in local, so in case of **ebsi** method is needed the **ESSIF** class to register it on the blockchain;

deleteDID : deletes a DID from the holder's wallet;

- * **Params**: the **DID** (string or object);
- * **Returns**: the result of the deletion, which is a **boolean**;

resolveDID : resolves a DID (in case of **ebsi** method, it is searched on-chain);

- * **Params**: the **DID** string;

* **Returns:** the resolved DID JSON with related metadata;

`importDID` : resolves and then imports a DID in the holder's wallet;

* **Params:** the DID string;

* **Returns:** the import result, which is a boolean;

`getCredentials` : gets the credentials owned by the holder;

* **Returns:** an array of credentials in JSON format;

`getCredential` : gets a specific credential owned by the holder;

* **Params:** the alias of the credential (e.g., `passport`);

* **Returns:** the credential in JSON format;

`getCredentialIDs` : gets the credential IDs (aliases) owned by the holder;

* **Returns:** an array of credential IDs (strings);

`storeCredential` : stores a credential in the holder's wallet;

* **Params:** the the credential's desired alias and the credential object;

* **Returns:** the storage result, which is a boolean;

`deleteCredential` : deletes a credential from the holder's wallet;

* **Params:** the alias of the credential;

* **Returns:** the deletion result, which is a boolean;

`presentCredentials` : returns the presentation of one or more VCs;

* **Params:** the `PresentationRequest` object;

* **Returns:** the `Presentation` in JSON format;

* **Notes:** the `PresentationRequest` object is a union of two types, `PresentCredentialsRequest` and `PresentCredentialIDsRequest`; in this way, a user can choose to pass the credentials or their IDs; other object fields are the `holderDID`, the `verifierDID`, and more;

* **Auditor:** this class serves as the verifier component in our SSI model. It implements four methods:

`getVerificationPolicies` : gets the verification policies, usable to verify the credentials;

* **Returns:** an array of JSON objects;

`verifyCredential` : verifies one or more VCs/VPs;

* **Params:** the `VerificationRequest` object, composed by two fields: `credentials` and `policies`;

- * **Returns:** the `VerificationResponse` object in JSON format (fields: `valid`, `results`);

`createDynamicVerificationPolicy`: creates a verification policy usable to verify the credentials;

- * **Params:** the desired `name`, the `DynamicPolicyArg` object, and two optionals: `update` (to make it updatable) and `downloadPolicy` (this depends by `DynamicPolicyArg`);
- * **Returns:** the creation result, which is a `boolean`;

`deleteDynamicVerificationPolicy`: deletes a verification policy (just if it has been created by the user);

- * **Params:** the `name` of the policy;
- * **Returns:** the deletion result, which is a `boolean`;

- * **ESSIF:** it enables the user to interact with EBSI blockchain (e.g., register a DID);

`onboard`: first step required for a DID registration on EBSI;

- * **Params:** the `bearerToken`, obtainable from the ebsi website, and the DID (`string` or `object`) to onboard;
- * **Returns:** the VC of the onboarding, or `false` if the onboarding fails;

`auth`: second step required for a DID registration on EBSI;

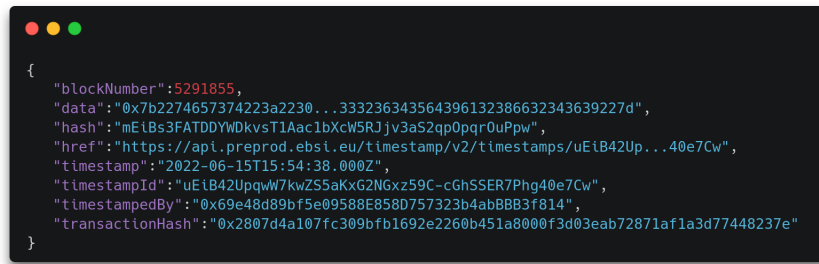
- * **Params:** the DID (`string` or `object`) to auth;
- * **Returns:** the authentication result, which is a `boolean`;

`registerDID`: last step required for a DID registration on EBSI;

- * **Params:** the DID (`string` or `object`) to register;
- * **Returns:** the registration result, which is a `boolean`;

`getTimestampByID`: gets the timestamp of a specific transaction;

- * **Params:** the `transactionID` (`string`);
- * **Returns:** the timestamp in JSON format;
- * **Notes:** a timestamp example can be seen in the [figure](#).



```

{
  "blockNumber": 5291855,
  "data": "0x7b2274657374223a2230...333236343564396132386632343639227d",
  "hash": "mEiBs3FATDDYWDkvsT1Aac1bXcW5RJjv3aS2qp0pqr0uPpw",
  "href": "https://api.preprod.ebsi.eu/timestamp/v2/timestamps/uEiB42Up...40e7Cw",
  "timestamp": "2022-06-15T15:54:38.000Z",
  "timestampId": "uEiB42UpqW7kwZS5aKxG2NGxz59C-cGhSSER7Phg40e7Cw",
  "timestampedBy": "0x69e48d89bf5e09588E8580757323b4ab8BB3f814",
  "transactionHash": "0x2807d4a107fc309bfb1692e2260b451a8000f3d03eab72871af1a3d77448237e"
}

```

Figure 3.5: EBSI transaction timestamp example

`getTimestampByTXHash`: gets the timestamp of a specific transaction;

- * **Params:** the txHash (string);
- * **Returns:** the timestamp in JSON format;

- * **Universal Resolver:** enables the resolution of DID registered in other blockchains than EBSI;

`resolveDID`: resolves a DID;

- * **Params:** the DID (string);
- * **Returns:** the resolved DID in JSON format;
- * **Notes:** to see the complete list of supported blockchains, refer to the [official repository](#).

- * **utils.ts:** in these files can be found utilities used in the SDK, but that can also be used by the users, such as:

- **types**, used to enhance the readability of the code, to avoid errors and to make the SDK easier to use;
- **consts**, especially for the API ports;
- **interfaces**, to define the structure of the objects used as input or output of the API calls, this way the user can easily understand the structure of the objects, and does not need to refer to the documentation each time;
- **functions**, such as `callAPI` (already [explained](#)) or `getId`, used by the SDK to get the ID of a key or DID.

- * **lib.ts:** here are placed functions that use multiple SDK methods also from different classes. Only two functions have been written so far, and almost only for testing purposes, but they can be used by the user as well, and there can be added more in the future if needed.

Problems and difficulties found. The most challenging thing about building the SDK was understanding everything about the SSI Kit. The developer needed to learn everything involved in the kit: the W3C standards for credentials issuing and verification, the basics of cryptography, API interaction, and more.

Frequently, the API calls' input or output parameters were not specified in the documentation (at least, the schema was specified but not the true meaning of specific fields). This way, the only manner to go on was to analyze the SSI Kit source code (this meant knowing a minimum of Kotlin) or contact the team on Slack.

Finally, a minor bug in the Custodian component was found during the SDK development. It has been reported, and the team fixed it in a week.

In the

Tests. The SDK's main components have been tested, focusing on the SSI Kit's four classes (Custodian, Signatory, Auditor, and ESSIF).

The final coverage is decent but is not 100%, so it could be slightly improved by testing untested branches.

In the underlying images, it can be seen (in this order) all files' final coverage, the statements coverage, and finally branches, functions, and lines coverage details.

All files

84.72% Statements **172/203** **64.97%** Branches **115/177** **91.48%** Functions **43/47** **84.02%** Lines **163/194**

Figure 3.6: All files' tests coverage percentages

| File | Statements | Branches | Functions | Lines |
|--------------|------------|----------|-----------|-------|
| Auditor.ts | 95.23% | 20/21 | | |
| Custodian.ts | 93.75% | 60/64 | | |
| ESSIF.ts | 100% | 17/17 | | |
| Signatory.ts | 100% | 14/14 | | |
| lib.ts | 65.51% | 19/29 | | |
| utils.ts | 72.41% | 42/58 | | |

Figure 3.7: Visual representation and statements coverage

| Branches | Functions | Lines |
|----------|-----------|--------|
| 68.75% | 11/16 | 100% |
| 75.9% | 63/83 | 95.23% |
| 63.63% | 14/22 | 100% |
| 72.22% | 13/18 | 100% |
| 25% | 1/4 | 50% |
| 38.23% | 13/34 | 80% |

Figure 3.8: Branches, functions, and lines coverage details

Documentation. The SDK documentation can be found at the link <https://matteocasonato.gitbook.io/ssikit-sdk/>. Here can be seen the specifications of the SDK classes and methods and some use examples.

3.2.3 Web Application Proof of Concept

After the SDK and smart contracts development, we merged the two macro components into a basic web application as a proof of concept of the final solution.

The application serves as an interface for smart contract and SDK interaction, enabling users to interact with verifiable credentials.

Structure

The whole application can be divided into two main components: the frontend, which contains the central part of the logic, thanks to React.js, and the backend, used just because some operations would not have been secure if done in the frontend, and something could not be done here because of compatibility issues (browser do not support all the cryptographic functions, as we explain later).

Frontend The frontend breaks down into six main pages:

- * **Holder:** provides a complete interface for the SDK **Custodian** class; this component is divided into three sub-components:

Keys: here the user can manage the keys used to generate the DIDs. The user can create a new key, export it, delete it and more;

DIDs: here the user can manage their DIDs. The user can create a new DID, load it, delete it;

Credentials: here the user can manage their credentials. The user can import a new credential, present it and delete it.

- * **Issuer:** provides a complete interface for the SDK **Signatory** class; this component is divided into two sub-components:

Issue: in this component, the user can issue a credential.

Revocations: in this component, the user can revoke a credential or check if a credential is revoked.

- * **Verifier:** provides a complete interface for the SDK **Auditor** class, and adds the on-chain functionalities for the verifiers; it is divided into two sub-components:

Verifications: here is possible to verify credentials and put the result on-chain as a verification record; also, verification records are searchable;

Verifiers: here the smart contracts owner can add new verifiers and search them on-chain;

- * **Contracts:** on this page, the smart contracts owner can register a new contract as trusted or untrusted, and search them on-chain;

- * **Diploma:** on this page, an hypothetical student can request a diploma request NFT, and then consume it to gain access to the final diploma certificate (VC);

- * **EBSI:** here an user can register a DID (generated with (ebsi) method) on the EBSI network; serves as an interface for the SDK **ESSIF** class.

Finally, the application provides the user a web3 wallet connector, so he can interact with the smart contracts where needed.

Backend We needed to add a backend to our application for two important reasons:

1. **Security:** the frontend is not secure enough to handle cryptographic operations, so we needed to move them to the backend; Specifically, when a verifier wants to add a new verification record, he must create a signature with the private key of its DID. The signature certifies that the real verifier is adding that record. As the application must interact with the user's private key, this must be done at the backend level: if the private key reaches the frontend, it is no longer secure. We avoided this by using the server as a REST API: when it has to sign, the user passes its private key's (public) ID. The backend creates the signature, and it is passed to the frontend. The signature can also be verified with another method.
2. **Compatibility:** the browser does not support all the cryptographic functions we need. In particular, we needed the elliptic curve Secp256k1, used for Ethereum accounts generation. The DID generated with the ebsi method must be created with a Secp256k1 key pair because when there is an interaction with the blockchain, the EBSI API will sign a message using the corresponding private key (which has to be compatible with an Ethereum account).

Assuming this, if we use cryptographic functions from the frontend (using React.js), the application will interact with the W3C Web Cryptography API. Unfortunately, this API does not support the Secp256k1 curve, so we have been forced to move this logic to the backend.

Functionalities

We now examine what the two main components of the web application, frontend, and backend, offer (obviously, the frontend offers functionalities to users, and the backend serves only the frontend).

Frontend. Now we show all the functionalities provided by each previously explained frontend's page.

Holder. This page is divided into three sections: **Keys**, **DIDs** and **Credentials**;

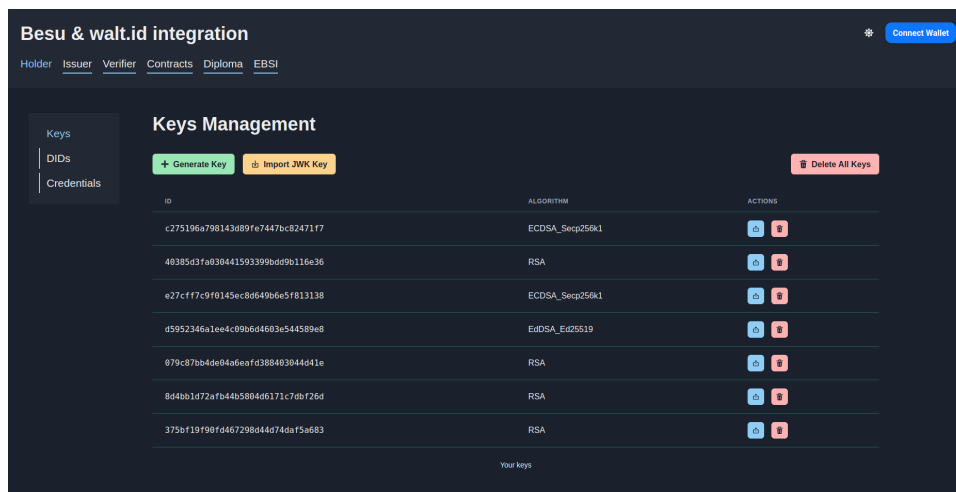


Figure 3.9: The Holder page (Keys section)

In the **Keys** section inside the **Holder** page, a user can:

- * Generate a key using a supported algorithm, by pressing the green button **Generate Key**;
- * Import a key in JWK format, yellow button;
- * Export a key (public or private) in the desired format, by pressing the blue button;
- * Delete a key, by pressing the red button;
- * Delete all the keys, by pressing the red button **Delete All Keys**;

In the **DIDs** section, a user can:

- * Create a DID using a supported method and a previously generated key, by pressing the button **Create DID**;
- * Import a DID inserting its string, by pressing the button **Import DID**;
- * Resolve a DID by pressing the button **Resolve DID** (remembering that **did:key** are in local, and **did:ebis** are resolved from the blockchain);
- * View a DID loading it from the local storage by pressing the button in the table;

- * Delete a DID (from local storage) by pressing the button in the table;
- * Delete all the DIDs, by pressing the button **Delete All DIDs**;

In the **Credentials** section, a user can:

- * Import a credential, giving it an alias, by pressing the button **Import Credential**;
- * Present one or more credentials, selecting them in the credentials table and by pressing the button **Present Credential**;
- * View a credential loading it from the local storage by pressing the button in the table;
- * Delete a credential by pressing the button in the table;
- * Delete all the credentials, by pressing the button **Delete All Credentials**;

Issuer. This page is divided into two sections: **Issue** and **Revocations**;

The screenshot shows the 'Besu & walt.id integration' interface. The top navigation bar includes 'Holder', 'Issuer', 'Verifier', 'Contracts', 'Diploma', and 'EBSI'. The 'Issuer' section is active, and the 'Issue' sub-section is selected. The 'Issue a credential' form contains the following fields:

- Select Template ID:** A dropdown menu with 'Verifiable' selected.
- Your DID:** A text input field containing 'did:key:z78op1_eGQK9dMGwo'.
- Subject DID:** A text input field containing 'did:example:123456789'.
- Select a Proof Type:** A dropdown menu with 'LD_PROOF' selected.
- Proof Config parameters:** A text area containing a JSON object with fields like 'verifierDid', 'issuerVerificationMethod', 'domain', 'nonce', 'proofPurpose', 'credentialId', 'issueDate', 'validDate', 'expirationDate', and 'dataProviderIdentifier'.

The **Selected template** section displays a JSON schema for a credential, including fields like '@context', 'credentialSchema', 'credentialSubject', 'currentAddress', 'dateOfBirth', 'familyName', 'givenName', 'gender', 'id', 'nameAndFamilyNameAtBirth', and 'personalIdentifier'.

The **Issued credential:** section shows the generated JSON credential, which is a copy of the schema with specific values for the 'credentialSubject' and 'id' fields.

Figure 3.10: The Issuer page (Issue section)

In the **Issue** section, an issuer can issue a credential by filling out the form and by pressing the button **Issue Credential**. First, he must select a template, used for the generation. Then, he must configure the credential, with the possibility of adding optional fields. The credential is then generated and inserted in the **Issued credential** field, and we can copy that and import locally the issued VC in the **Holder** page. In a real use case, the credential would be sent to the holder using secure communication protocols such as **OIDC**.

In the **Revocations** section, an issuer can revoke a credential by inserting the corresponding private revocation token he generated when he issued the credential. The revocation is off-chain, and we have not yet implemented the on-chain call (from the

frontend) that would change the verification record (however, the function is implemented in the smart contract). Also, anyone can check if the credential has been revoked by inserting the corresponding public revocation token.

Verifier. This page is divided into two sections: **Verifications** and **Verifiers**;

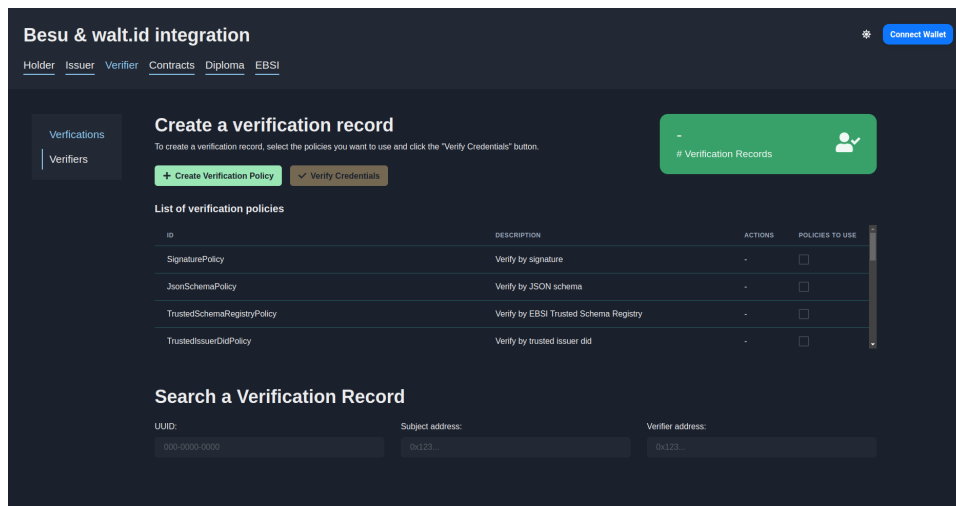


Figure 3.11: The Verifier page (Verifications section)

In the **Verifications** section, a verifier can:

- * Create a verification policy by pressing the green button;
- * Verify a credential and register the result on-chain as a verification record. To do so, he must select at least a verification policy from the list. Before adding the result on-chain, the verifier creates a signature with the private key of its DID. This way, anyone can resolve their DID and use the public key (findable in the credential's `verificationMethod` field) to verify the signature;
- * Search for an on-chain verification record.

In the **Verifiers** section, the contract owner can register on-chain a new verifier, adding it to the trusted verifiers' list. Anyone here can see the list of trusted verifiers and search for a specific one. Here is where we manage to merge the off-chain and on-chain solutions in the PoC. Other functionalities can be added (e.g., on-chain revocation after off-chain happened), but for time reasons, we have only implemented the verification record registration.

Contracts. In the underlying image can be seen the page screenshot.

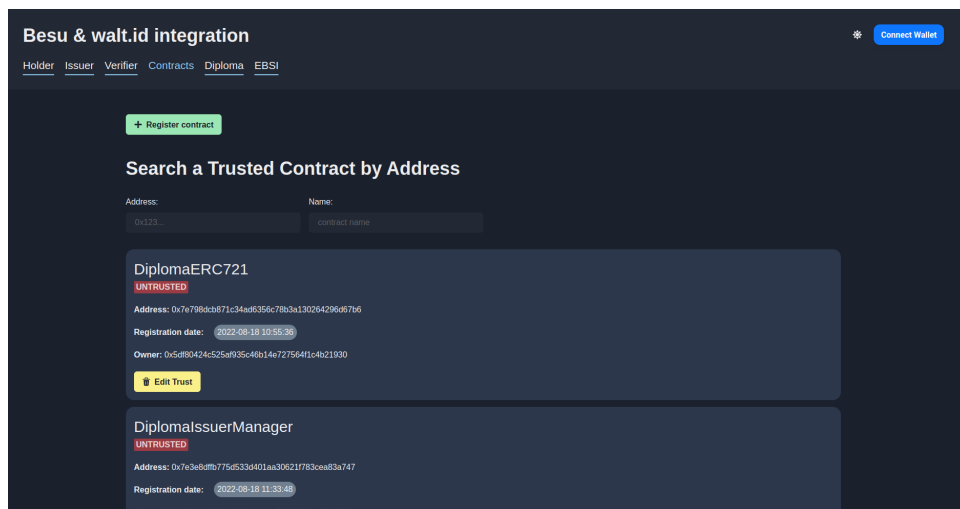


Figure 3.12: The Contracts page

On this page, the contract owner can register a contract as trusted or untrusted in the on-chain list. Anyone can see the list of contracts and search for a specific one.

Diploma. In the underlying image can be seen the page screenshot.

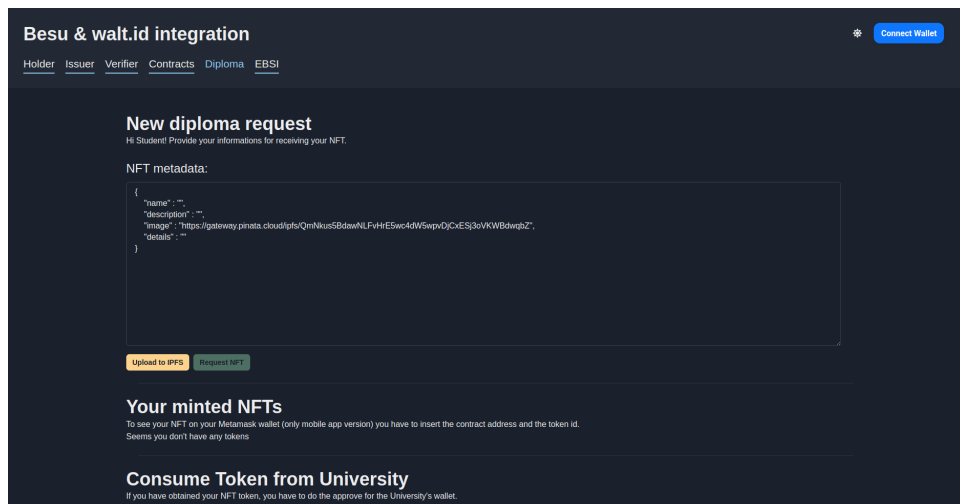


Figure 3.13: The Diploma page

On this page⁴:

- * A student can request an NFT usable by the user to officially request the diploma;
- * The university sees all the NFT requests and can accept them by minting the Request NFT to the student's wallet;
- * A student can approve the university to burn the Request NFT; after that, the university can consume (or burn) the Request NFT and issue the VC diploma to the student (issuing procedure is not implemented).

EBSI. In the underlying image can be seen the page screenshot.

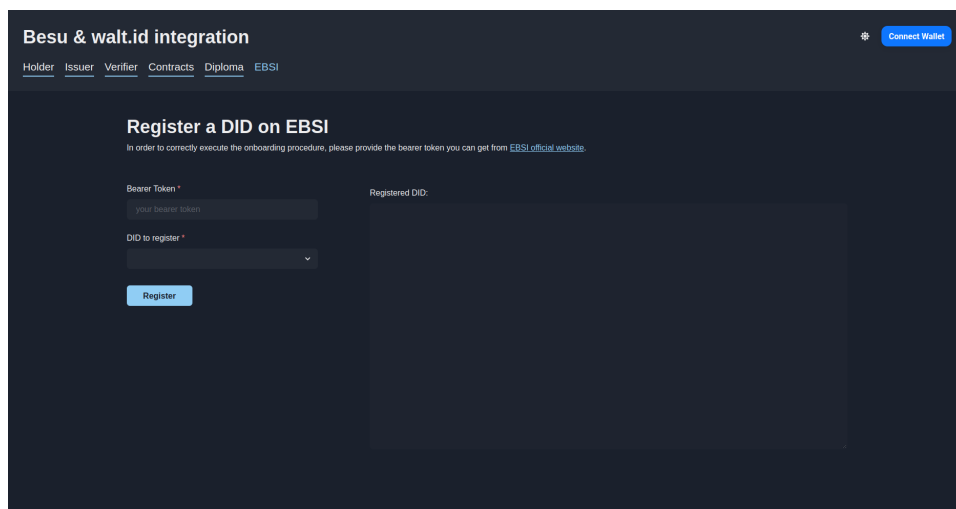


Figure 3.14: The EBSI page

A user can register a DID on the EBSI network on this page. The user must insert the `bearerToken`, obtainable on the EBSI official website. The interface simplifies the registration process: as explained in the SDK section, the process needs three steps, but the user sees only one.

⁴The details of this whole operation are not explained in this paper, as they are examination subjects of another student's thesis.

Backend. The backend is implemented as a REST API. We used it to implement the creation and verification of the signatures made by the verifiers when they register on-chain a verification record.

The functions made available to the frontend are two:

- * `/createSignature`: the verifier signs the verification record with its DID private key. In the request body, the verifier must insert the `JSON signatureRequest` object, which has two fields: `keyId`, where is specified the private key (public) ID, and `message`, which is the message to sign (in this case, the verification record); the call response is the signature in `JWS` format;
- * `/verifySignature`: any user (from the frontend, if the interface is implemented) can verify a signature. In the request body, the verifier must insert the `JSON verificationRequest` object, which has three fields: `verifierDid`, where specified the DID of the verifier who signed the message, and `message`, which is the message to verify (also in this case, the verification record: if the verification record data coincides with the decoded signature payload, then the signature is valid and the verification record is valid too); if the signature is valid, the call response is the signature payload (which should be equal to the verification record), otherwise the function will throw an error.

Problems and difficulties found

Generally, we have not found crucial problems during the PoC development. The only difficulty found was the signature creation; it took a while for us to find the real problem (i.e., the `Secp256k1` curve incompatibility with the W3C Web Cryptography API). Also, we had to compromise: verifiers, to generate the signature, can only use DID created with the `ebsi` method (so also with `Secp256k1` curve). The `/verifySignature` has to be revisited to enhance interoperability with other methods and encryption algorithms.

3.3 Discussion

3.3.1 Achievements

3.3.2 Acquired knowledge

3.3.3 Future developments

3.3.4 Personal evaluation

Conclusion

Bibliography

Bibliographical references

James P. Womack, Daniel T. Jones. *Lean Thinking, Second Edition*. Simon & Schuster, Inc., 2010.

Websites consulted

Manifesto Agile. URL: <http://agilemanifesto.org/iso/it/>.