Technische Hochschule Brandenburg
University of
Applied Sciences

Self-sovereign Identity: Development of an Implementation-based Evaluation Framework for Verifiable Credential SDKs

Brandenburg University of Applied Sciences
Department of Economics

Master's Thesis

submitted by

Philipp Bolte

September 25, 2021

First supervisor:	Prof. Dr. rer. nat. Vera G. Meister
Second supervisor:	Jonas Jetschni, M.Sc.

Statutory Declaration

I hereby attest that I have written this thesis independently without any outside help and that I have used only the sources cited.

Brandenburg, September 25, 2021

Philipp Bolte

Abstract

Diese LaTeX-Vorlage ist für Berichte, Bachelor- sowie Masterarbeiten gedacht. Natürlich ist sie nicht perfekt und jede Art der Verbesserung wird dankend angenommen. Bei Fragen zur Verwendung oder Anregungen zur Verbesserung können Sie mir diese gern an markus.brandt1992@gmail.com senden.

An entsprechender Stelle werden Beispiele für die Verwendung von Abkürzungen, Zitaten, Abbildungen, Tabellen und die Einbettung von Code gegeben.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

CA	Certificate Authority
DID	Decentralized Identifier
IDP	Identity Provider
IIW	Internet Identity Workshop
SDK	Software Development Kit
SSO	Single sign-on
SSI	Self-sovereign Identity
VC	Verifiable Credential
VP	Verifiable Presentation

1 Introduction

The Internet has become a cornerstone of coexistence in today's world. With over 4.66 billion Internet users worldwide [Jo21], it determines how we communicate, think, inform ourselves, and interact with one another. As a result, huge networks of people are being created in which different cultures are coming closer together and knowledge is being shared like never before. A central enabler for the functioning of such a digitalized society are digital identities [Li20].

Over the course of our lives, we generate a large amount of digital identities from a wide variety of services, including Facebook, Twitter, WhatsApp, GitHub, LinkedIn, and many more. They represent us in this digital realm, are part of our personality, and allow us to identify ourselves online. Because of the way we manage digital identities in the current era, users mostly own separate identities for each service or go through centralized, federated identity providers like Google or Facebook. As a result of these key developments, silos of identity data emerged, which are problematic concerning efficiency, security, and privacy. This creates a dependency towards the services that have full control over the identity data. This makes it difficult for users to control how services exploit this power for their own interests. In addition, various data leaks and hacks in which sensitive user data became public show that the current approaches are not suitable for the problems of these modern times [Sw21]. [Eh21, pp. 2-3]

In contrast, the SSI paradigm takes a new approach by giving users full control of their digital identities through various novel approaches [FCA19, p. 103059]. This work examines this new approach from a developer's point of view to test its practical applicability. In the next sections, the scope, related work and the research approach will be discussed.

1.1 Scope of Work

For a successful realization of Self-sovereign Identity (SSI) concepts, the existence of good solutions for developers is critical. This ensures that the barriers to a successful adoption of SSI are kept to a minimum, simplifying and speeding up the entire process. A good toolset and developer experience is thus a key enabler for SSI.

With this in mind, an overview of the most important solutions[1] in the SSI space is

[1] synonymous to Software Development Kits (SDKs), libraries, frameworks, and platforms

established throughout the thesis. To scope the work accordingly, this work looks at the solutions in terms of how closely they can map the lifecycle of a Verifiable Credential (VC). This decision was made due to VCs being a key artefact in SSI as they hold the actual verifiable data, e.g. vaccination status or birthdate, of a subject [SLC19]. The overview is intended to serve as an entry point for developers to get a general view of the capabilities of existing solutions and to give starting points for further research.

Furthermore, a use case agnostic reference implementation is presented that implements four of the presented solutions based on the lifecycle. It can serve developers as a basis for their own work, but above all enables practical validation and the gathering of experience during its development. This way, the knowledge gained flows directly into a new evaluation framework, which, in addition to other software selection frameworks, can provide concrete help in selecting the most suitable solution from the developer's point of view. In addition, it can reveal shortcomings in current solutions that need to be addressed for successful adoption of SSI in practical use cases. So the objective of this work, besides the scientific contributions, is to generate added value for the whole ecosystem.

1.2 Related Work

At the current time, there does not appear to be any comparable work that addresses the topic in a manner corresponding to section 1.1. The most similar is [NJ20] who have developed a mobile wallet based on uPort that covers login, VC issuance as well as verification. Based on the experience gained, an evaluation of uPort has been made as well. However, uPort is currently no longer being developed, and the assessment is also based on only a fraction of the VC lifecycle and basic principles for SSI.

Another paper by [Ku20] defines a comprehensive evaluation framework from an enterprise perspective that, compared to other papers, also covers aspects such as user experience, technology and compliance. It is characterized by a wide range of questions that are used for the evaluation of 43 solutions. However, the list of solutions considered is outdated and missing important players (see e.g. MATTR and Trinsic). Furthermore, the assessment does not provide any practical guidance for developers. A clear analysis of the SSI-relevant features, e.g. with regard to the VC lifecycle, does not exist.

Otherwise, many papers seem to focus on theoretical foundations or evaluation of existing solution based on two things: (i) architecture [GMM18] concerning privacy [Be19], performance [Bo20], use case [Ku20], various variations [Al16, Sp21, AL20, Bo20, FCA19, Ca05] of SSI principles [Bo19, Bo20, DT20, DP18, FCA19, SNE20], and (ii) the interoperability between those systems [Ho20, Jo20]. This clearly shows that there is a deficit in terms of works that look at existing solutions based on their practical features and applicability from a developer's point of view. This thesis addresses some of these gaps and thus clearly contributes to the field of research.

1.3 Methodology

The process for achieving the objectives from section 1.1 can be divided into four basic steps: gain theoretical foundation, create solutions overview, develop reference implementation, and define the evaluation framework (see figure 1.1).

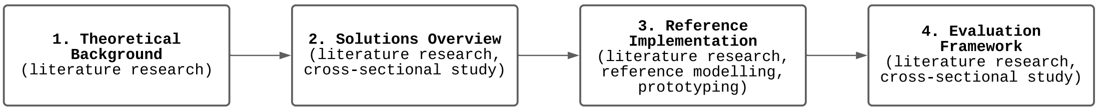

1. Theoretical Background
(literature research)
2. Solutions Overview
(literature research, cross-sectional study)
3. Reference Implementation
(literature research, reference modelling, prototyping)
4. Evaluation Framework
(literature research, cross-sectional study)

Figure 1.1: Research Approach

For this purpose, various methods of business and information systems engineering according to [WH07] are applied. Through a literature research, an overview of existing papers and books is created, which on the one hand serves for building a theoretical foundation, but also represents the basis for all other steps.

To find the literature, keywords such as "Self-sovereign Identity" and the following complex query based on [Bo19] were used at Google Scholar:

```
("Self-sovereign identity" OR "Self sovereign identity ")
OR (
    ("block-chain" OR "blockchain") AND ("identity management")
    AND ("solution" OR "implementation" OR "review" OR "survey")
    AND ("verifiable credentials" OR "decentralized identifiers")
)
```

Furthermore, the method of cross-sectional studies, mainly in the form of expert questionnaires, is used. A clearly defined set of questions is used to validate the solutions overview, but also to identify evaluation criteria for the evaluation framework. More details on the selection of experts and the questions are described in section X.X. For the creation of the reference implementation, the methods of reference modelling and prototyping are used. In combination, they allow the development of a software prototype that represents a particular problem in a simplified way and whose analysis can contribute to the discovery of new knowledge. This is especially interesting for the development of the evaluation framework. Moreover, it should also be noted that this thesis follows the general idea of generating real-world artefacts defined by [He07] as part of design science research.

To conclude the methodology, a combination of research approach and applied methods defined previously is reflected in the following research questions:

1. What libraries, platforms, or SDKs are available for implementing Verifiable Credentials?
2. Which SDKs do experts from the field recommend using?
3. Which criteria for evaluating Verifiable Credential SDKs can be derived after developing a reference implementation?

2 Self-sovereign Identity

Health care, social security, education, access to financial services — this is just a small list of requirements that are essential for a decent life and are usually taken for granted by people in the Western world. Yet, there are more than 1.1 billion people worldwide who cannot provide identification and thus cannot access the most basic services. Digital identities could make a significant contribution towards solving this problem and giving people the chance to participate in society on a more equal playing field. [Wo17]

As already mentioned in chapter 1, however, current implementations of such digital identities are insufficient for the problems of our modern times. [SNA21] divided these problems into four categories:

1. Data Ownership and Governance
2. Password-Based Authentication
3. Fragmented Identity Data
4. Data Breaches and Identity Fraud

The former describes the fact that users have no ownership over their digital identities and thus cannot exercise any control over them. Service providers often take advantage of this and use collected data to create comprehensive profiles of their users and thus sell tailored advertising space on corresponding marketplaces for high figures. The lack of control also means that service providers can temporarily or permanently deny users access to their digital identity at any time. At the beginning of 2021, this led to much discussion as the account of former U.S. President Donald Trump was permanently banned from Twitter. One of the central concerns was whether service providers have too much power over users' liberties [No21]. In addition, given the frequent and often repeated use of weak passwords, the heavy reliance on password-based authentication is a security risk that may lead to identity theft. If users want to protect themselves, they need to use different and complex passwords for each of their accounts, which quickly becomes a complicated undertaking without a password manager. A study by the password manager LastPass, for example, found that a business customer manages an average of 191 passwords [St17]. While the use of such tools greatly simplifies the management of passwords, they too can pose a major security risk and do not completely protect the user [OR20, Or21, To21]. Alternatives such as Single sign-on (SSO), where users authenticate to other service providers using for example their Google account, can solve this problem but lead to even greater dependency and centralization. The third issue involves identity data being spread across a large set of service providers, making it difficult to maintain. As a result, duplicates, errors, and outdated data sets are common. The lack of open standards also complicates interoperability between providers, which could theoretically be used to retrieve, move, or delete personal data. Efforts like the Data Transfer Project founded by Microsoft, Google, Twitter, and Facebook try to simplify the transfer of data between providers, but after more than 3 years show very few actual successes [Mi20, Ho21a, Lo20] and are being criticized for pushing small competitors even further behind [BC18, p. 15]. [SNA21, pp. 2-3]

One of the biggest problems, however, are data breaches. In June 2021 alone, there were 235 breaches with 1.16 billion stolen records, with a total of 18.9 billion records stolen in 1,785 breaches in the first half of 2021 [Ri21c]. Looking at the past, there have been quite a few major hacks [Sw21], including:

- Yahoo (2013): 3 billion accounts
- Marriott (2018): 500 million customer records
- Alibaba (2019): 1.1 billion entries
- LinkedIn (2021): 700 million accounts

A survey of 413 people by [Ma21o] found that 73% of participants had been affected by at least one, but an average of 5.3 data breaches. In addition, the majority blamed themselves for the breaches, with only 14% aware that service providers were responsible.

These are decades-old problems that were already critically discussed by Kim Cameron in 2005. Cameron, who last worked as Chief Architect of Identity at Microsoft from 1999 to 2019, wrote the following on a blog article [Ca05]:

> *"The Internet was built without a way to know who and what you are connecting to. This limits what we can do with it and exposes us to growing dangers. If we do nothing, we will face rapidly proliferating episodes of theft and deception that will cumulatively erode public trust in the Internet."*

Cameron attributes these problems to the lack of an identity layer on the Internet, which has resulted in many services having to find their own solutions. He calls this a *patchwork of identity one-offs*, which fundamentally still exists today and is difficult to resolve. The reason for this, he says, is a lack of consensus and an unwillingness to give up too much control over identity data. A solution for this is, according to him, an *identity metasystem* that abstracts away deeper complexities similar to hardware drivers or TCP/IP and only loosely couples digital identities to the systems. Such an open identity layer could only be successful if it fulfilled the seven laws of identity defined by Cameron. These include criteria such as user control, consent, pluralism and minimal disclosure. [Ca05]

Over the years, these ideas, among others like [Ma12, id14, Al16], gave rise to the concept of Self-sovereign Identity (SSI). It is intended to eliminate the shortcomings of today's established concepts by placing the users in the center and giving them back complete control over their identity data. A user can decide what, to whom and how much data is shared without being dependent on a central authority. The emergence of blockchain technology and various new standards in recent years gave a new boost to implement SSI in reality. [St21, pp. 6-7; To17, pp. 8-9]

SSI is an entirely new approach to digital identities on the Internet and is seen as a paradigm shift that deeply affects the infrastructure and power distribution of the Internet [PR21, p. 3]. For a more profound look at the topic, this chapter takes a closer look at Self-sovereign Identity. To do so, the concept of identity and the different types of identities will be discussed first. This is followed by a historical look at the different stages of digital identities, taking a closer look at the previous concepts of SSI. After a basic foundation has been built, standards that have been established in recent years and are intended to make SSI feasible in reality are described. Finally, the SSI architecture with its components and roles will be looked at.

2.1 Identity

What defines a human being? One would probably get various answers to this question, such as its name, gender, place of residence, profession, hobbies, religion, charitable activities, party affiliation or even a combination of all these characteristics. [CK01, p. 206] describes in his work that a person's identity is not just a single, fixed construct, but consists of several partial identities. Thus, depending on the context in which a person finds itself, it takes on one of its various partial identities, which represents it as a human being more or less. For example, a partial identity for health care consists of its medical history, while the partial identity towards work contains received certificates. Nevertheless, these different parts of the identity are not necessarily considered separately, as they can also overlap in certain aspects of information. It is important to mention that a person decides which information to share at which time towards which entity. In figure 2.1 the concept of partial identities is illustrated exemplarily by a person Alice.

An important balancing act is to disclose the right amount of data to maintain anonymity, but also to provide the other person with the necessary information. For the purchase of a water, the kiosk vendor should not ask for any personal data, whereas verification of age when buying alcohol is a valid reason for information disclosure. In reality, official documents, such as state identification documents, or sometimes unofficial documents, such as customer cards, are usually used for such proof of identity. Here, users have full control over their documents as they are under their control, and only they can decide self-sovereignly whom and when to show them. Official identification documents are also produced and standardized to ensure the highest possible level of security and interoperability. Other countries can verify such documents without explicitly contacting authorities, simply by looking at the document. Confidence in the validity of the data arises from the fact that the verifying party trusts the authority issuing the document. [St21, p. 6]

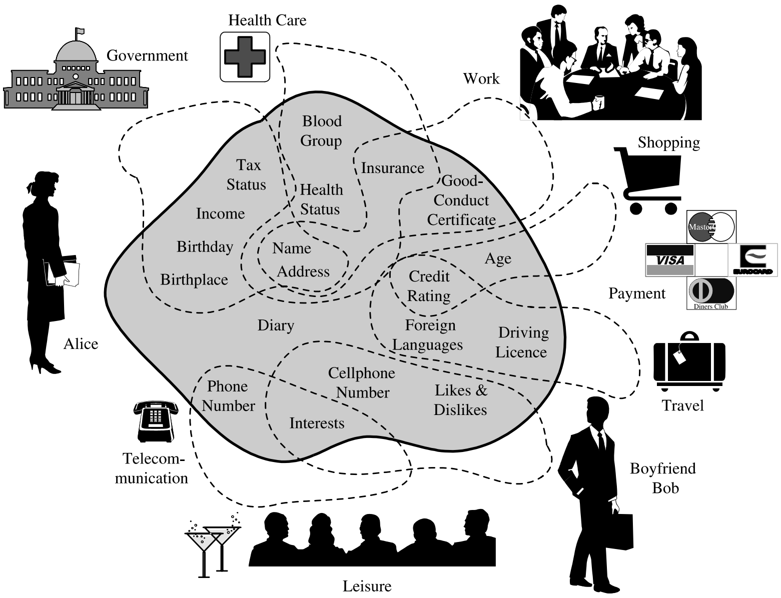


Figure 2.1: Partial Identities of Alice extracted from [CK01]

As a result of the increasing digitalization of various branches of life, many processes are shifting to the digital world. Digital identities, which are similar to analogue identities in terms of their basic idea, are now being used for interacting with digital services. They allow entities, such as people or objects, to authenticate themselves online through certain attributes and thus prove their identity [MG20, p. 103; Bu20]. A more precise definition is given by Cameron [Ca05], who defines digital identity as *"A set of assertions that a digital subject makes about itself or another digital subject"*. In this context, a digital subject is *"A person or thing represented or existing in the digital realm which is being described or dealt with."* and the attributes mentioned can be represented in the form of claims, which are defined as *"An assertion of the truth of something, typically one which is disputed or doubted"*. The problem is that analogue identities and their documents usually have no or not widely accepted [Kr19, Ko21] digital representations that could be used as a digital identity. From this emerged the patchwork of identity one-offs described in chapter 2, resulting in a divergence of digital identities from their original counterparts concerning their characteristics. To better understand this development, the next section describes the different stages of digital identities in more detail. [St21, p. 10; Eh21, p. 2]

2.2 Stages

As indicated in the last section, Allen's work [Al16] has had a major influence on what is today considered Self-sovereign Identity and has been cited in over 100 works according to Google Scholar. According to him, digital identities, or online identities, have gone through four major stages since the beginning of the internet. These are examined in more detail below and show which developments led to the emergence of SSI.

2.2.1 Centralized Identity

Centralized identities are identities that are issued and verified by a single party or hierarchy. The oldest examples of this are IANA (1988) for the administration of IP addresses, ICANN (1998) for domain names and Certificate Authorities (CAs), which play a major role today, particularly in connection with SSL certificates. Especially with the latter, the hierarchical structure of CAs becomes obvious when one looks at an SSL certificate in the browser. Here, a root authority allows another organization to manage its own hierarchy, while at all times the root authority has full control. This is highly critical for numerous reasons. For example, one entity has complete control over identities and can delete them at any time or even issue false identities. The latter can happen both willingly and unwillingly as a result of a hack. Due to the centralized nature of such authorities, they and thus also the complete hierarchy (chain of trust) are targets of attack, which has been shown in recent years [Bo12]. Just like these organizations, due to the lack of an identity layer, all services on the internet developed similar centralized solutions (see chapter 2. This manifests itself above all in the various accounts that an internet user has to manage for various services. Again, users have little control over their data. [Al16]

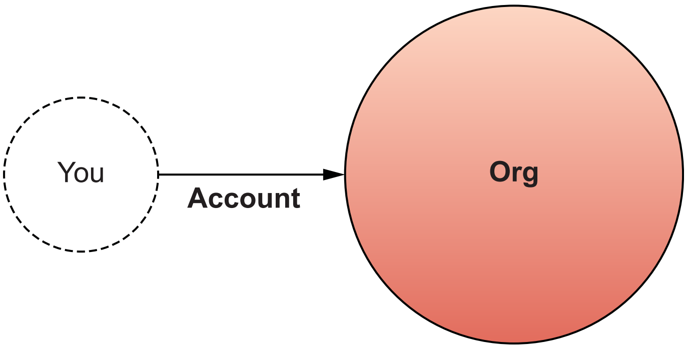


Figure 2.2: Relationship in centralized identities extracted from [PR21, p. 7]

In addition, the user has to manage the abundance of login credentials efficiently and securely. However, it is also a challenge for the services, as they have to store a large amount of sensitive data securely and in compliance with data protection laws. Nevertheless, the beneficiaries here are the services, as they can act flexibly and independently of third parties and have full control over the data. [Eh21, p. 6]

2.2.2 Federated Identity

The second stage of development is represented by the so-called federated identities, which were intended to break down the hierarchies based on a single authority. Here, various commercial organizations developed a model in which control was to be divided between several federated authorities. One of the first projects in this area was Microsoft's Passport in 1999, where Microsoft created a single, federated identity for users that could be used on multiple sites. However, this unification came with the price that Microsoft was now at the center of the federation and could thus exert full control. Other efforts, such as Liberty Alliance Project, founded in 2001, attempted to create an actual federation between multiple companies in which control was distributed among them. The result, however, was a kind of oligarchy in which users still had no control over their data. In the end, the sites remained authorities. [Al16]

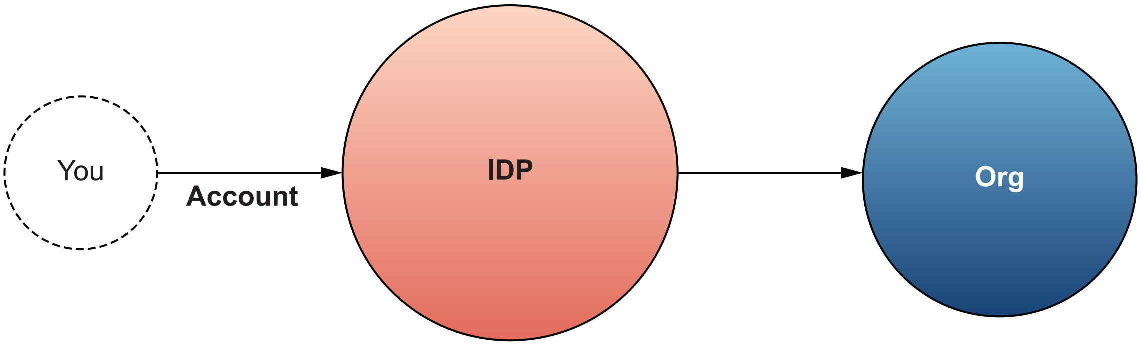


Figure 2.3: Relationships in federated identities extracted from [PR21, p. 8]

Nevertheless, this type of digital identity is advantageous in that users do not have to manage an identity/ account for each service and companies have less administrative effort. The identity provider, e.g., Microsoft, acts as the issuer and owner of the data and is thus the central point of contact if a user wants to log on to another service of the federation. The user therefore has no control over his data and is dependent on the continued existence of the identity provider. Due to the abundance of sensitive data, it is possible for the identity provider Identity Provider (IDP) to aggregate information from various areas in order to create user profiles, which in itself can lead to various problems. [Eh21, pp. 6 - 7]

2.2.3 User-Centric Identity

The goal of user-centric identity is to make federations obsolete and allow the individual to assert control over their identities across multiple authorities [Al16]. The foundations for this, according to [Al16], lie in [JHF03], in which a *persistent online identity* to be integrated directly into the architecture of the Internet was proposed, making federations unnecessary. One of their central demands was that users should have the right to control their own digital identity. This includes, among other things, the ability to decide what information is collected as part of their digital identity and who has access to which parts. Earlier approaches such as Microsoft's Passport or the Liberty Alliance Project were unable to meet these requirements because, as stated by [JHF03], they were too business-oriented and thus too focused on the privatization of information. According to them, everyone's digital identities should be a public good that should not be tied to the financial interests of a private company, as their commercial interest may not overlap with those of society.

These thoughts were guiding and influenced various future organizations and initiatives. One influential organization in this area has been the Internet Identity Workshop (IIW), which grew out of efforts by the Identity Commons and the Identity Gang. The IIW community played a major role in shaping what is understood by user-centric identity and supported key standards such as OpenID (2005), OpenID 2.0 (2006), OAuth (2010), and OpenID Connect (2014). [Al16] summarizes the focus of these efforts with the terms user consent and interoperability, which were non-existent or difficult to implement in previous models. These protocols have also been able to achieve significant success when considering the abundance of social logins from for example Facebook, Google, GitHub and Microsoft, which have taken a central position on various websites [PR21, p. 8]. Nevertheless, the original approach of user-centric identities could not be realized further. Like in previous approaches, the identity data and thus absolute control remain with the SSO providers who register them. [Al16] mentions OpenID as an example, which theoretically allows users to set up their own OpenID providers. However, the complexity is so great that in reality this option is hardly ever used. Accordingly, the original problems that user-centric identities were supposed to solve could only be partially solved, since central, mostly private actors have maintained their authority over identity data. Fundamentally, user-centric identities are still federated identities that are now merely interoperable, which is why some literature [Eh21, PR21] does not list them separately. [Al16]

2.2.4 Self-sovereign Identity

[Al16] refers to Self-sovereign Identity as the next and most current stage of digital identities, which is intended to solve the issues of all previous stages. In contrast to user-centric identities, users are not only at the center of the identity process, but should also be able to completely own and manage their identities. [PR21, p. 12] describes this as a "[...] shift in control from the centers of the network [...] to the edges of the network [...]", according to which all users interact directly with each other in a self-sovereign manner as peers. This evolution can be seen in figure 2.4.

Apparent here is the new element *registry*, which is used as a (decentralized) public key infrastructure [PR21, p. 89]. A more detailed explanation of this is given in section 2.3. To further describe the character of SSI, [Al16] defined 10 principles, with which he connects to previous works like the "Laws of Identity" by [Ca05]. These are:

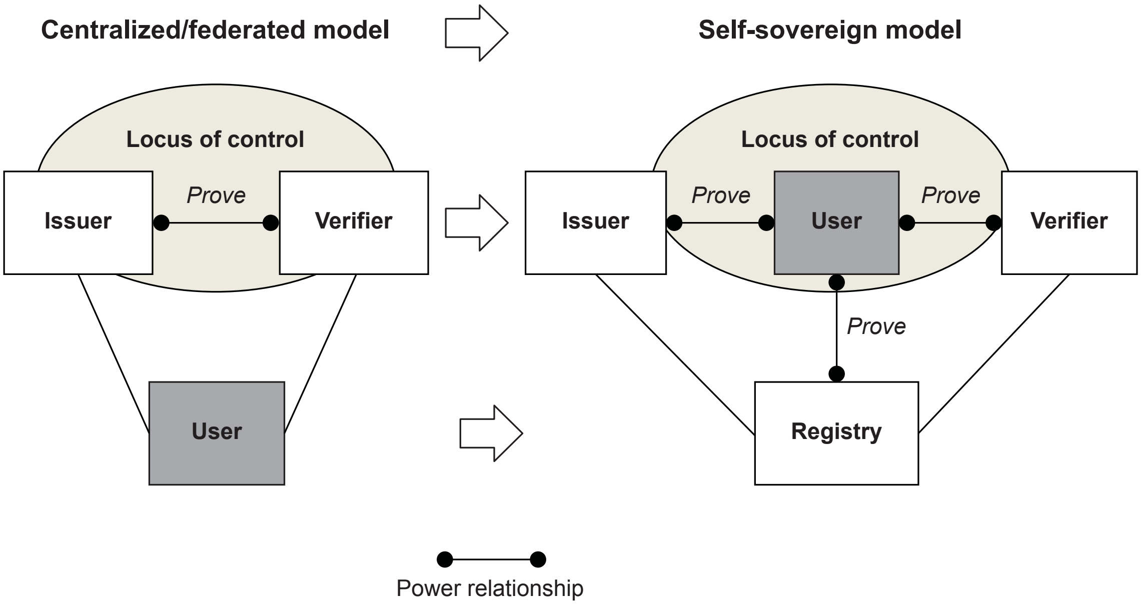


Figure 2.4: Shift of control with SSI extracted from [PR21, p. 12]

1. Existence: “Users must have an independent existence.”
2. Control: “Users must control their identities.”
3. Access: “Users must have access to their own data.”
4. Transparency: “Systems and algorithms must be transparent.”
5. Persistence: “Identities must be long-lived.”
6. Portability: “Information and services about identity must be transportable.”
7. Interoperability: “Identities should be as widely usable as possible.”
8. Consent: “Users must agree to the use of their identity.”
9. Minimalization: “Disclosure of claims must be minimized.”
10. Protection: “The rights of users must be protected.”

SSI, according to [Al16], has its origins in the term “Sovereign Source Authority”, which originated in [Ma12]. In this work, Marlinspike attributes to every human being the right to an identity, which is hindered by tight state structures. In the same year, work began on the Open Mustard Seed by Patric Deegan, which was intended to give users control over their digital identity in a decentralized system. This later resulted in the Windhover Principles (2014), under which the term Self-sovereign Identity appeared [id14, Hu14]. These state, among other things, the following: [Al16]

> *“Individuals [...] should have control over their digital identities and personal data ensuring trust in our communications, and the integrity of the data we share and transact with. [...] Individuals, not social networks,*

> *governments, or corporations, should control their identity credentials and personal data."*

Over the course of the following years, SSI in connection with blockchain technology was frequently also being discussed in the IIW community as well, and various ideas were being developed. This eventually led to some official agencies taking a closer look at this topic. For example, the U.S. Department of Homeland Security Science & Technology division published a report in 2015 in which it addressed the previously discussed topics by the IIW. The EU and countries such as China and Korea have also recognized the potential. In order to make SSI implementable in reality, various new standards have been defined over the years in the W3C, among others, which will be discussed in more detail in the next section. [PR21, p. 6]

2.3 Standards

In this section, the two most important standards *Decentralized Identifier (DID)* and *Verifiable Credential (VC)* will be discussed in more detail, as they are the basis for SSI and various subsequent standards.

2.3.1 Decentralized Identifier

Throughout history, humans have built up various imaginary networks in which they have to identify and address themselves or objects. Be it physical, mutual networks, in which one identifies itself with one's name or be it postal or telephone networks in which it is the addresses or the telephone numbers. In the age of the Internet, various others have been introduced, such as IP addresses, e-mail addresses, domain names or usernames in social networks. Consequently, there are a large number of identifier systems, which can vary greatly in their nature and place of application. Zooko Wilcox-O'Hearn published an article on this subject in 2001 [WO01], in which he describes a trilemma in identifier systems. According to this, an identifier can probably have at most two of the following properties: [PR21, pp. 183-186]

1. Human-readable: Identifiers have semantic meaning in human language and thus have low entropy.
2. Secure: Identifiers are unique and thus bound to a single entity. Spoofing and impersonation should not be possible as well.
3. Distributed: The namespace of the identifier is not managed by any central authority. Identifiers can be generated and resolved independently.

According to this, e-mail addresses, domain names and user names, for example, are human-readable and secure, but do not fulfil the distributed criterion. However, this is precisely what is needed for an SSI ecosystem in which users can own and manage their identity in a self-determined and sovereign manner. With the advent of blockchain technology, however, first solutions emerged that sought to break this problem and thus Zooko's Triangle. This involved the concept of decentralized domain services (see e.g. Namecoin or ENS), with which human-readable, secure and distributed identifiers can be generated. However, these are usually tied to the underlying blockchain and have intrinsic value due to their human-readable nature, which is why domains/ identifiers are often registered and held in such systems without actual use [Ka15], questioning the utility of the system.

To meet the requirements related to identifiers in an SSI system, the W3C standard *Decentralized Identifier* has been defined. These are *globally unique* and *location-independent* identifiers that can be generated autonomously by entities without central authorities and provide the ability to prove control over them through cryptographic evidence. Regarding Zooko's Triangle, DIDs don't attempt to be human-readable and thus satisfy the properties secure and distributed [PR21, p. 185]. [Sp21]

The characteristics of a DID can be summarized in the following points [PR21, p. 160]:

1. Persistent: DIDs have no intrinsic expiration date and do not need to change.
2. Resolvable: DIDs are resolvable to retrieve additional metadata.
3. Cryptographically Verifiable: The owner of a DID can cryptographically prove control over it at any time. This is enabled by a public and private key pair being assigned to a DID.
4. Decentralized: A DID *can* be issued/ generated independently of a central authority.

To better visualize how DIDs work, figure 2.5 provides the DID architecture with all its components and relations.

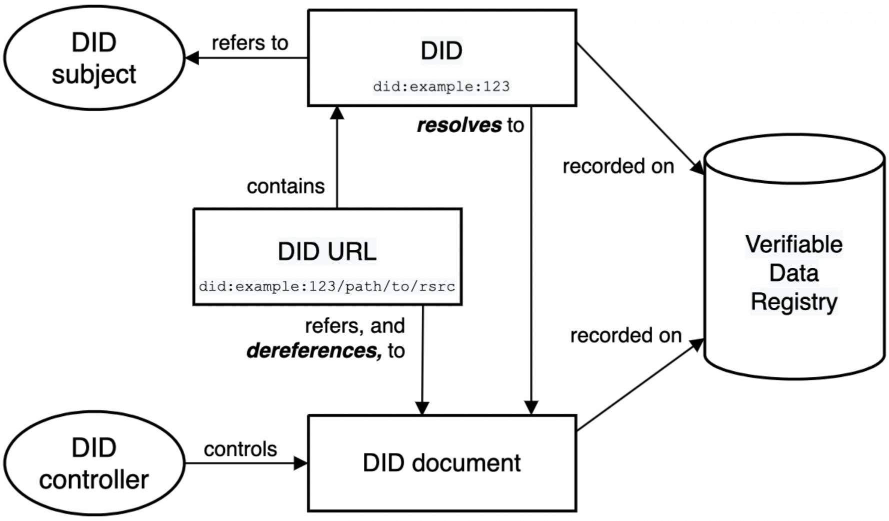


Figure 2.5: DID architecture extracted from [Sp21] (TODO: VECTOR!)

At the top is the DID subject, which can be any entity and is represented by the DID. The DID itself, for example `did:example:123456789abcdefghi`, is the actual identifier, which consists of three parts. The first part `did` describes the identifier schema, `example` the DID method and the third part `123456789abcdefghi` a DID method-specific identifier which can be used to resolve the DID document according to the DID method. Since DIDs are location-independent, they *can* be recorded in different Verifiable Data Registries, for example blockchains or decentralized file systems. The DID method defines any mechanisms for creating, resolving, updating and deactivating DIDs and their DID document that may be recorded on a specific data registry. Examples of DID methods that rely on blockchains or layers built on top of them as a Verifiable Data Registry include `did:ethr`, `did:btcr`, and `did:ion` which are in contrast to static DID methods like `did:key` which do not require a Verifiable Data Registry and usually wrap the public key from which the DID document can be derived [PR21, p. 171]. The DID document contains various metadata about the associated DID and define things like verification methods, public keys, and possible service endpoints for interactions with the subject. Additionally, paths can be attached to DIDs to address specific resources within the DID document. These are the so-called DID URLs. An example DID document can be found in listing 2.1. [Sp21]

```
{
    "@context": [
      "https://www.w3.org/ns/did/v1",
      "https://w3id.org/security/suites/ed25519-2020/v1"
    ]
    "id": "did:example:123456789abcdefghi",
    "authentication": [{
      "id": "did:example:123456789abcdefghi#keys-1",
      "type": "Ed25519VerificationKey2020",
      "controller": "did:example:123456789abcdefghi",
      "publicKeyMultibase": "
   zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
    }]
}
```

Listing 2.1: DID document example extracted from [Sp21]

Lastly, figure 2.5 includes the DID controller. This is another entity that is authorized to make changes to the DID document. Most of the time the DID controller is also the DID subject, but in some cases these entities can be different (see for example parent-child relationship). [Sp21]

In conclusion, the DID specification is a W3C standard that enables decentralized identifiers for SSI ecosystems. The next section presents Verifiable Credentials, which, in conjunction with DIDs, form the basic building blocks of SSI.

2.3.2 Verifiable Credentials

Now that a standard has been created with DIDs, with which entities can independently generate unique and authority-independent identifiers, there is still a need for a data model with which, in combination with DIDs, identity data can be represented in a standardized way. For this purpose, the W3C defined the Verifiable Credential (VC) standard. This defines VCs as a set of claims stated by an issuer in a tamper-evident way, allowing integrity and authorship to be cryptographically verified. A claim is thereby defined, similarly to subsection 2.1 by Cameron, as an *"[...] assertion made about a subject."* where a subject is a *"[...] thing about which claims are made."*. A VC is written in JSON-LD and consists of three basic components, which are visualized in figure 2.6. [SLC19]

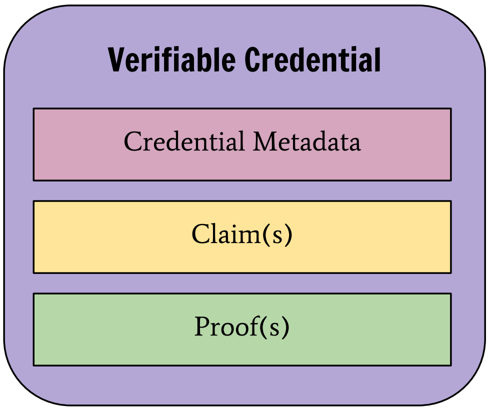


Figure 2.6: Components of VC data model extracted from [SLC19]

The credential metadata can define various properties of the VC including its issuer, an expiration date, credential types, or a revocation mechanism that can be used to check whether the issuer has revoked the credential. Thereafter, a set of claims can be defined by the issuer, which contains statements about the subject of the VC. Finally, cryptographic proofs can be attached and used to verify the validity of the credential's contents. The standard distinguishes between two types of proofs: [SLC19]

1. External Proof: The contents of the credential are wrapped, thus converted into a different, cryptographically verifiable format. A well-known example of this are JSON web tokens, which are used today in many identity systems for the transfer of claims between multiple parties, providing a certain compatibility to existing systems. Effectively, the set of claims is represented in a digital signature, the JSON web signature. Since these were developed for the JSON format, proofs can only refer to an entire credential and not to individual attribute sets [He20b]. [SLC19]

2. Embedded Proof: The proof is contained in the data and is therefore JOSN-LD native, which makes pre- or post-processing of the data unnecessary [SLC19]. Such so-called Linked Data Proofs use Linked Data Signatures, which can

create proof chains on the basis of the semantic structure of JSON-LD. This enables proofing on an attribute basis, rather than per credential as in external proofs. This high amount of flexibility also creates room for other technological possibilities, such as zero knowledge proofs. [He20b]

In listing 2.2 is an exemplary JSON-LD document, which is leveraging the Verifiable Credentials Data Model, attesting the credential subject a bachelors degree.

```
{
   "@context":[
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
   ],
   "type":[
       "VerifiableCredential",
       "UniversityDegreeCredential"
    ],
    "issuer":{
        "id":"did:key:z6MkhMLpju5tqtbd54BSv7Sq2oRWQo6n..."
    },
    "issuanceDate":"2021—05—26T08:33:40.681Z",
    "credentialSubject":{
       "id":"did:key:z6MkhMLpju5tqtbd54BSv7Sq2oRWQo6n...",
       "type":"BachelorDegree",
       "name":"Bachelor of Science and Arts"
    },
    "proof":{
        "type":"Ed25519Signature2018",
        "created":"2021—05—26T08:33:40Z",
        "jws":"eyJhbGciOiJFZERTQSIsImI2NCI6ZmFsc2UsImNyaXQ...",
        "proofPurpose":"assertionMethod",
        "verificationMethod":"did:key:
   z6MkhMLpju5tqtbd54BSv7Sq2oRWQo6njMEywrbWAGAp3442#z6MkhM..."
    }
}
```

Listing 2.2: Example of a Bachelors degree as a Verifiable Credential

Listing 2.2 also shows the basic building blocks described earlier. The document starts with a context definition to reference the semantic vocabulary. This is followed by a definition of the credential types, the issuer and the issuing time (credentials metadata). After that follows the actual claim in the object `credentialSubject`, where the subject and the corresponding degree are defined. At this point it also becomes clear how the DID and VC specifications are intertwined: both issuer and subject are defined by their DID. The public private key pair belonging the issuer's

DID becomes relevant especially in the next point: the proof. Here, the issuer uses the private key coupled to its DID to generate the Linked Data signature and thus makes the credential verifiable.

Another important part of the standard are Verifiable Presentations (VPs). If a holder of a Verifiable Credential wants to present it to someone, it can combine one or more VCs in a Verifiable Presentation without invalidating the cryptographic proofs. This approach has several advantages. On the one hand, the owner of the credential can specify granularly which credentials it wants to disclose and, at the same time, a type of proof of ownership can be provided. This becomes particularly clear if one considers the structure of such a VP in figure 2.7. [SLC19]

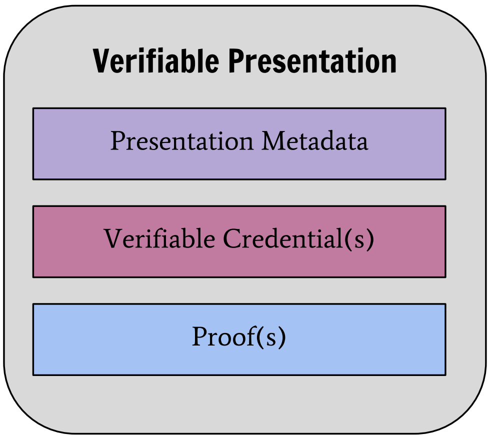


Figure 2.7: Components of a Verifiable Presentation extracted from [SLC19]

Once again, metadata is defined at the beginning, which can include attributes like the context and types. This is followed by the VCs to be presented, which are listed directly one after the other without any changes. Finally, a cryptographic proof follows, which the owner of the credentials generates with the private key of its DID. This protects the integrity of the presentation and at the same time certifies that the credentials are actually presented by the owner of the DID. The inclusion of the attributes `challenge` and `domain` in the proof can also provide protection against replay attacks, in which an attacker presents the intercepted presentation again to another verifier without authorization. [SLC19]

Having introduced Decentralized Identifiers and Verifiable Credentials as the backbone of an SSI ecosystem, the next section will specify a few things in more detail.

2.4 Architecture

In this section, various functional aspects of SSI are described in more detail. For this purpose, the roles and interactions in a SSI system are described, followed by a general look at the technology stack.

2.4.1 Roles

In an SSI ecosystem, there are three basic roles that participants can occupy: issuer, verifier, and holder. They have already been briefly described in subsection 2.3.2 and are therefore an integral part of the VC standard. The three roles are briefly presented below: [PR21, pp. 25-26; SLC19]

1. *Issuer*: An entity that makes statements within a VC about a subject. Such an entity can be organizations such as governments, universities, but also private individuals or objects such as sensors. An issuer transmits VCs to holders.
2. *Holder*: An entity that requests or receives VCs from issuers and manages them in a credential repository/ digital wallet. However, a holder may not always be the (credential) subject. Examples of these cases include a parent (holder) holding VCs for its child (subject) or a friend (holder) filling a prescription at the pharmacy for its sick friend (subject). Holders can also generate Presentations from Verifiable Credentials and show them to a verifier.
3. *Verifier*: An entity that wants to verify certain attributes or claims of a subject. It may receive these in the form of VP, which may contain those claims from one or more VCs. However, holders have control at all times over which attributes are passed to the verifier.

The roles and their relations are often called the trust triangle, as it describes how trust is formed in an SSI ecosystem. Like in the real world, trust in the credentials comes from a verifier trusting the issuer. The figure 2.8 shows this triangle, but also visualizes how the roles interact with VCs, which is why this process is also called the Verifiable Credential Lifecycle. [PR21, pp. 25-26; SLC19]

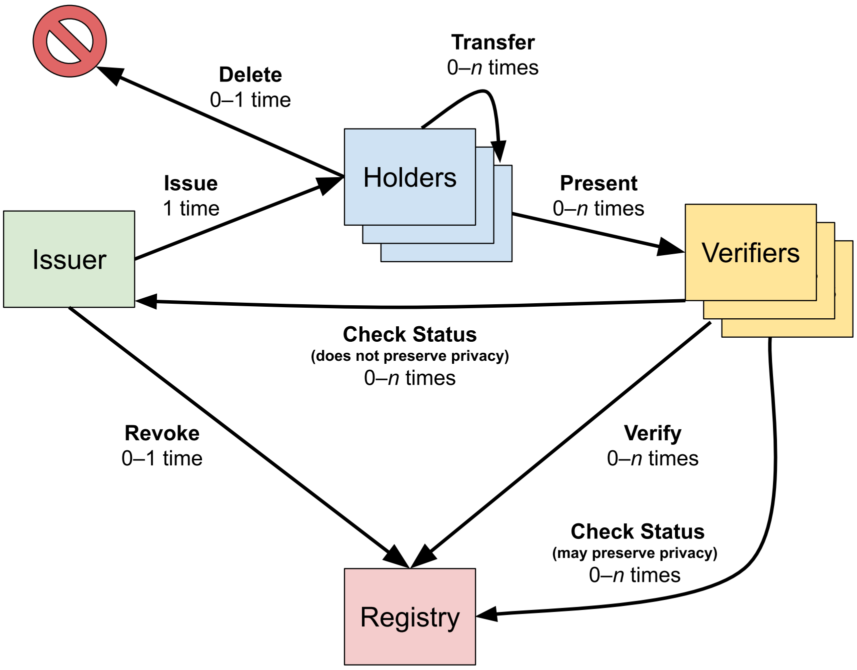


Figure 2.8: Verifiable Credential Lifecycle edited and extracted from [SLC19]

The lifecycle shows which phases a Verifiable Credential goes through and which roles perform which actions in these phases. At this point, the process is described using a Verifiable Credential representing a bachelor's degree as an example. Here, the issuer is a university, the holder is an alumnus, and the verifier is a potential employer. The process is as follows: [SLC19]

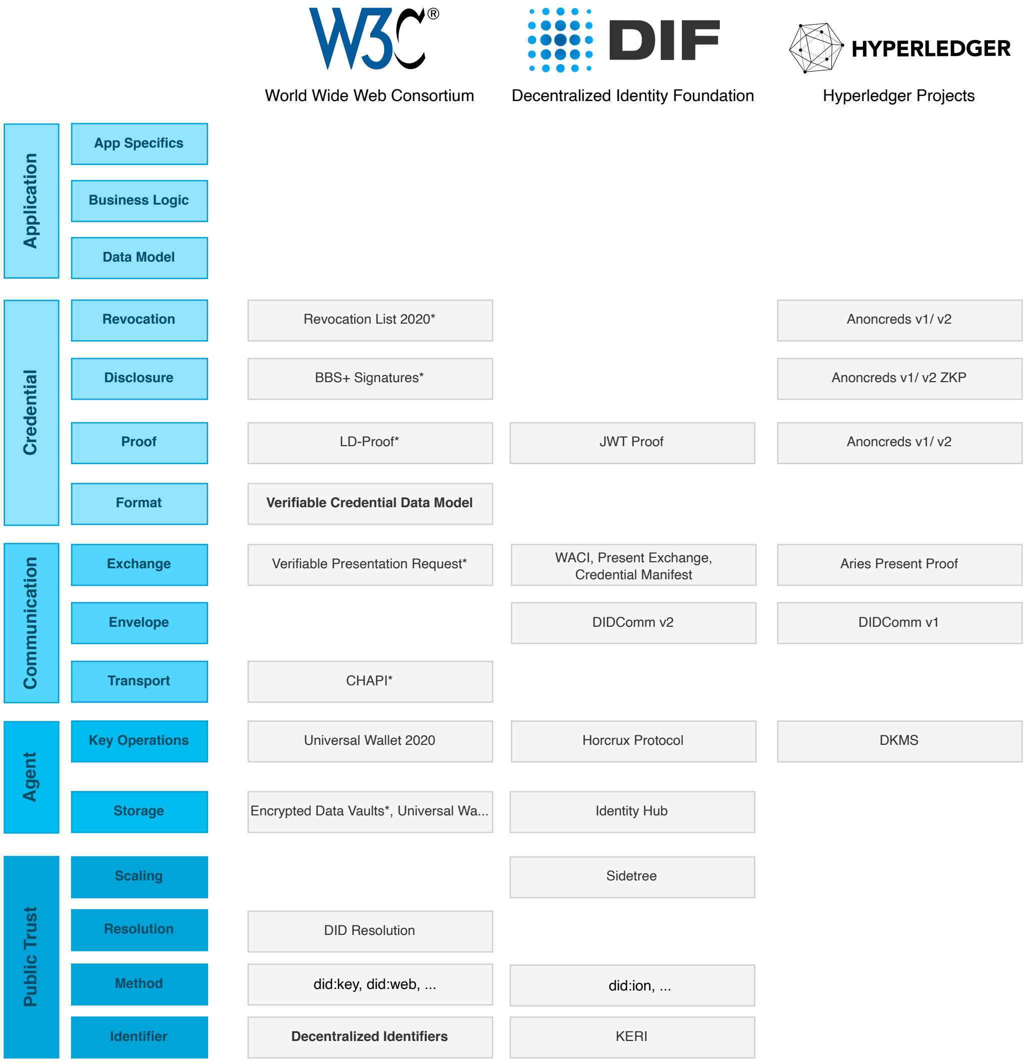
1. *Issue*: The now alumnus has successfully defended its thesis. Its university then issues a Verifiable Credential to the DID of the alumnus with its own DID. As holder and subject, the alumnus stores the VC in its digital wallet.
2. *Transfer*: The alumnus can transfer this VC to another holder, e.g., if he authorizes a friend to go to a governmental authority for him with the degree.
3. *Present*: The alumnus presents the VC of the bachelor's degree, optionally inside a VP, to the potential employer as part of his application in order to have his degree verified.
4. *Verify & Check Status*: The potential employer checks the authenticity of the credential. This includes firstly checking that the credential meets the standard, the proofs are valid and is not revoked. To check the proof, the employer must resolve the DID documents of the DIDs in the credential to obtain the public keys. For this, depending on the DID methods used, the employer may need to query a verifiable data registry (see subsection 2.3.1.
5. *Revoke*: If the university wants to revoke and thus invalidate a VC for some reason, it can do so. Depending on the implementation, this is also done with some kind of decentralized or central registry. One of them is described later on in subsection 2.5.3.
6. *Delete*: A holder can delete a VC from its digital wallet at any time, which does not affect its overall validity.

This process can be applied to any other use case and creates a system through defined standards, technologies and the described trust model, which in its basic characteristics also takes place in real interactions and where trust can form between entities.

2.4.2 Technology Stack

Looking at the SSI technology stack, figure 2.9 provides an overview that divides it into five basic layers. This is based on previous work from the Decentralized Identity Foundation and the Trust over IP organization [He20a, Yi21, Da21]. A notable change is that here the communication layer has been broken out of the agent layer. Even though it is mostly used by agents, it's not part of the agents itself, but rather an implemented software module leveraging the communication layer.

The public trust layer is the baseline layer and thus forms the basis for all other layers above it. The aim here is to create a public trust registry that includes, for



		World Wide Web Consortium	Decentralized Identity Foundation	Hyperledger Projects
Application	App Specifics			
	Business Logic			
	Data Model			
Credential	Revocation	Revocation List 2020*		Anoncreds v1/ v2
	Disclosure	BBS+ Signatures*		Anoncreds v1/ v2 ZKP
	Proof	LD-Proof*	JWT Proof	Anoncreds v1/ v2
	Format	**Verifiable Credential Data Model**		
Communication	Exchange	Verifiable Presentation Request*	WACI, Present Exchange, Credential Manifest	Aries Present Proof
	Envelope		DIDComm v2	DIDComm v1
	Transport	CHAPI*		
Agent	Key Operations	Universal Wallet 2020	Horcrux Protocol	DKMS
	Storage	Encrypted Data Vaults*, Universal Wa...	Identity Hub	
Public Trust	Scaling		Sidetree	
	Resolution	DID Resolution		
	Method	did:key, did:web, ...	did:ion, ...	
	Identifier	**Decentralized Identifiers**	KERI	

* Unofficial draft by W3C Credentials Community Group

Figure 2.9: SSI technology stack, standards, and efforts based on [He20a, Yi21, Da21]

example, DIDs and their DID methods and thus serves as a (decentralized) public key infrastructure. As already mentioned, this may or may not imply the need for technologies like blockchains or decentralized file systems. In the subsequent agent layer, which fundamentally allows an entity to *"[...] take actions, perform communications, store information, and track usage of the digital wallet."* [PR21, p. 192] and thus handles tasks related to storing VCs and DIDs and performing key operations. This includes for example the generation of proofs, but also the deactivation or generation of new key pairs. On top of this is the communication layer, which handles the communication between agents. This includes transport, envelope, and credential exchange standards and protocols. On the fourth level is the credential layer, which includes all standards and technology used in the credentials' data model, such as formats, types of proofs, disclosure, and revocation. The top level is the application layer, which creates user applications based on the underlying layers that cover and implement specific use cases. This includes concrete data models for the credentials, but also business and application-specific logic and technology. [He20a, Yi21, Da21, PR21]

In addition to the general structure and elements of the layers, figure 2.9 contains concrete standards and community efforts which provide the layers with a technological basis. A differentiation is made here between the concrete efforts of the three most important organizations in this area. It should be noted at this point that this overview does not claim to be complete and is merely illustrative.

2.5 Recent Developments

As mentioned in the last subsection and in figure 2.9, there are various community efforts that try to fill the gaps in the SSI stack. Therefore, three of the most important efforts are examined in more detail in the following subsections. The selection was made by observations of the reference implementation and the last Internet Identity Workshop in April 2021.

2.5.1 DIDComm

DIDComm, or DID communication, is a standard for secure, asynchronous, peer-to-peer communication between agents based on the DID standard. It is managed by the DID-Comm Working Group of the Decentralized Identity Foundation [Ha21] and originated from efforts of the Hyperledger project [Ha19]. Messages are agnostic of the transport medium, so existing protocols such as HTTP, Bluetooth, NFC, or even QR codes can be used. Moreover, the standard focuses on machine-readable messages, which enables a broader mass of use cases where all kinds of entities can exchange any kind of encrypted messages. [PR21, pp. 96-97]

If an entity A wants to send a message to entity B, it prepares a JSON message and retrieves the DID document of B's DID. Out of this, it needs two pieces of information: A messaging endpoint and the public key. With the latter, A can encrypt the message so that only B can decrypt it. In addition, A attaches a signature that it created with its own private key. This allows B to verify the integrity and origin of the message. Depending on the messaging endpoint and the transport route, the message is either delivered directly or scheduled for several hops via intermediaries. The standard provides various routing-specific information for this. If B now receives the message, it can decrypt it with its private key and verify the signature with As public key. If everything is correct, B can reply in the analogous way to As's procedure. The specification describes its goals in eight points: [Ha21]

1. Secure: Temper-proof using cryptography.
2. Private: Intermediaries don't know who is when communicating about what. Senders can be anonymous.

3. Decentralized: Trust is based on keys derived from control of DIDs
4. Transport-agnostic: Usable with any transport protocol. No matter if simplex, duplex, synchronous, asynchronous, online, or offline.
5. Routable: Messages can be routed like email through any kind of infrastructure.
6. Interoperable: Independent of hardware or software.
7. Extensible: Easily extensible by developers.
8. Efficient: Low resource requirements.

To better understand the structure of a DIDcomm message, listing 2.3 shows a plaintext version.

```
{
    "typ": "application/didcomm-plain+json",
    "id": "1234567890",
    "type": "<message-type-uri>",
    "from": "did:example:alice",
    "to": ["did:example:bob"],
    "created_time": 1516269022,
    "expires_time": 1516385931,
    "body": {
      "messagespecificattribute": "and its value"
    }
}
```

Listing 2.3: Plaintext DIDComm message extracted from [Ha21]

Subsequently, `type` describes the media type of the message, i.e. whether it is unencrypted, encrypted, and or signed. This is relevant for the corresponding library how to handle the content. DIDComm relies here on some JSON Web Algorithms from the JOSE (Javascript Object Signing and Encryption) family, which standardizes these cryptographic operations. `id` is the message ID and identifies the message exactly. Next, `type` describes what kind of message is in plaintext so that it can be handled correctly at the application level later. The next two attributes `from` and `to` define the DID of the sender and the DIDs of the recipients, followed by timestamps defining the creation and expiration time of the message. All attributes up to this point are the so-called message header, which is followed by the `body` containing the actual message. [Ha21]

The specification is much more detailed in many points, but this will not be considered further here with regard to the scope of this work. DIDComm as a secure communication method over DIDs is considered one of the most promising specifications in this area [PR21, p. 97] and can significantly contribute to how entities can exchange simple messages or even VCs peer-to-peer. Nevertheless, DIDComm is still relatively new, so its toolset and adoption is still relatively small.

2.5.2 BBS+

With regard to subsection 2.2.4, Allen describes in his ten principles for SSI, among other things, the principle of “minimization”, according to which the number of released claims should be kept as low as possible. This is also known as selective disclosures, according to which a user can keep certain attributes of a credential secret, which corresponds to the blackening of documents in the analog world. The next step to this approach are so-called zero knowledge proofs, where the actual attribute is not shared, but an assertion of a value that confirms what the verifier wants to know (predicates). For example, if an age check takes place, the verifier does not actually need to know the exact age but only the fact whether the person is over 18 or not. There are two basic approaches to this in the community: Camenisch-Lysyanskaya signatures and BBS+ signatures. Camenisch-Lysyanskaya signatures were one of the first implementations in the form of Anoncreds v1, but were dependent on published schemes for each credential on a ledger and therefore not standard-compliant. Furthermore, they were accompanied by large keys and large credentials that were costly to generate [Zu21]. [Yo21, pp. 17-18]

In the fall of 2020, MATTR announced BBS+ LD proofs that promised the same benefits while maintaining compatibility with the VC specification and reducing credentials and signature size. In addition, signatures were significantly faster to generate and the dependency on a ledger was not needed any more. [Zu21]

Table 2.1: Comparison of BBS+ and CL signatures (based on MA20b, He20c)

Domain	Criterion	BBS+ Signatures[1]	CL Signatures
Size	Private Key	32 Bytes	256 Bytes
	Public Key	96 Bytes	$771 + \frac{257}{msg}$ Bytes
	Signature	112 Bytes	672 Bytes
	Proof	$368 + \frac{32}{hidden\_msg}$ Bytes	$696 + \frac{74}{msg}$ Bytes
Performance	Key Generation	1 ms	8.8 sec
	Signing[2]	2.58 ms	93 ms
	Verifying[2]	5.23 ms	11 ms
	Proof Generation[2]	10.6 ms	13 ms

[1] BLS12-381 elliptic curve
[2] For 10 messages

Technically, BBS+ LD proofs rely on a combination of Linked Data proofs, the JSON-LD credential schema and BBS+ signatures. This combination allows generating proofs for presentations that can contain only a subset of attributes of the original VCs without affecting the semantic expressiveness and cryptographic integrity. Attribute filtering is performed using JSON-LD framing, while BBS+ is used to generate a so-called multi-message signature. As the name suggests, instead of one signature of one message, here a signature is composed of an array of messages. This ultimately allows the resulting signature to be derived by the holder so that it can only represent a part of the attributes. Unlike Camenisch-Lysyanskaya signatures, current implementations of BBS+ do not yet allow the use of predicates to enable comprehensive zero knowledge proofs. Cryptographically, however, such support is possible, but has not been the focus of efforts to date. Alternatively, the values of such predicates can be incorporated directly by the issuer as a separate attribute in the VC, which can then be disclosed by the holder through selective disclosures. The associated specification "BBS+ Signatures 2020" [LS21b] is currently an unofficial draft, which is managed by the W3C Credentials Community Group. [Yo21, pp. 18-20]

2.5.3 RevocationList2020

An elementary part of the VC lifecycle is the revocation step (see subsection 2.4.1). It allows issuers of VCs to revoke them at any time due to various reasons. Such reasons may be, for example, the expiration of a credential whose expiration date was not known at the time of issuance (e.g., office building access card), or incorrectly issued credentials (e.g., fraud).

The VC standard describes this process step and some important requirements, but does not define a standard for it. It only provides that there could be so-called "revocation registries" which could be part of the verifiable data registry and some privacy considerations concerning data leakage and correlation [SLC19]. Since such a registry can be located at any storage location, as already described in subsection 2.3.1 and 2.3.2, the writer thinks that a distinction could be made at this point between on-chain (on a blockchain) and off-chain (off a blockchain) storage solutions for revocation. An example of the former is Anoncreds v1, which implements such a registry on-chain that can be used to retrieve values used for the revocation process [Ha18, Hy18]. In contrast to this are off-chain solutions such as the Revocation List 2020 specification (see figure 2.10), which is managed by the W3C Credentials Community Group and currently an unofficial draft [LS21a].

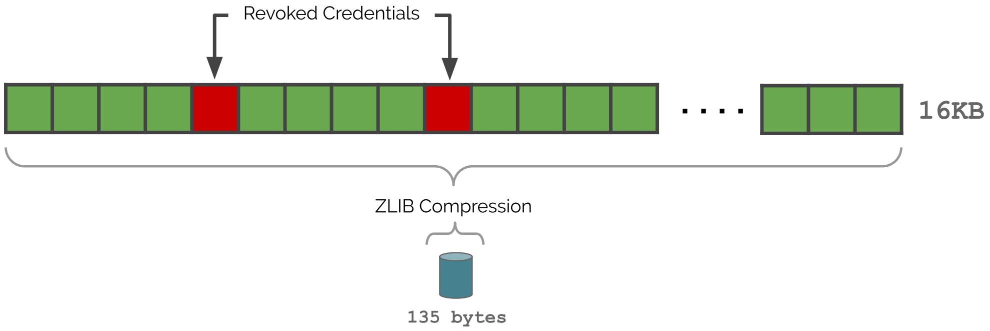


Figure 2.10: Workings of Revocation List 2020 (extracted from [LS21a])

It is a revocation mechanism based on a compressed bitstring that is compatible with existing architectures on the Internet. The issuer of VCs maintains and publishes its own revocation list (bitstring) for its credentials, which contains their status. Each credential is assigned a position in the list whose bit value describes its status (1 = revoked, 0 = not revoked). The specification advises a default bit string size of 16 KB, which can then hold the status of 131,072 credentials. However, since in most cases the majority of credentials has not been revoked and therefore have the value 0, the size can be reduced to a few hundred bytes using compression methods such as ZLIB. If a verifier now wants to retrieve the status of a credential, it retrieves the revocation list from the issuer, which contains the status of over 100,000 credentials. It can decompress this and check the position from the credential in the bit string to check the revocation status. The advantages of this approach can be summarized as follows: [LS21a]

- Efficient: The compression of the bit string uses little storage space and bandwidth.

- Privacy: The status of a credential can be hidden among several others. This makes correlation difficult for issuers, since an entire list is retrieved at any time, instead of the state of a single credential. If higher privacy is needed, the length of the bit string can be increased accordingly. Furthermore, only a minimum of information is published, so no data is published outside of an anonymous status.

- Compatibility: The list can be hosted by an issuer directly or distributed through a content distribution network, which further increases privacy since the issuer no longer handles requests directly.

For its implementation, the issuer must publish the compressed bitstring in a Verifiable Credential signed by itself. The data model can be seen exemplary in listing 2.4.

```
{
    "@context": [...],
    "id": "https://example.com/credentials/status/3",
    "type": ["VerifiableCredential", "
   RevocationList2020Credential"],
    "issuer": "did:example:12345",
    "credentialSubject": {
      "id": "https://dmv.example.gov/credentials/status/3#94567",
      "type": "RevocationList2020",
      "encodedList": "H4sIAAAAAAAAA-3
   BMQEAAADCoPVPbQsvoAAAAAAAAAAAAAAA..."
    },
    "proof": { ... }
}
```

Listing 2.4: Example RevocationList2020 credentials (edited and extracted from [LS21a])

The specification requires that this credential must contain a `id` that references the id within the credential whose status is being represented. In addition, the type `RevocationList2020Credential` must be included, as well as the type and encoded list under `credentialSubject`. Correspondingly, the specification also describes how the published revocation list can be correctly referenced as a revocation method in an VC issued by the issuer. This is illustrated in listing 2.5.

```
{
    "@context": [...],
    "id": "https://example.com/credentials/23894672394",
    "type": ["VerifiableCredential"],
    "issuer": "did:example:12345",
    "credentialStatus": {
      "id": "https://dmv.example.gov/credentials/status/3#94567"
      "type": "RevocationList2020Status",
      "revocationListIndex": "94567",
      "revocationListCredential": "https://example.com/credentia
    },
    "credentialSubject": {
      "id": "did:example:6789",
       "type": "Person"
    },
    "proof": { ... }
}
```

Listing 2.5: Example VC referencing a RevocationList2020 credential (edited and extracted from [LS21a])

In 2.5, the object `credentialStatus` is the primary object to be considered. In this, the `id` is found in the first place, which corresponds to the id in the `credentialSubject` of the RevocationList2020 credential. The `type` attribute clearly indicates that this credential uses the RevocationList2020 revocation method, followed by the index of the revocation status of the credential in the list (`revocationListIndex`) and a URL leading directly to the VC containing the encoded revocation list (`revocationListCredential`). [LS21a]

RevocationList2020 is a promising approach to implement privacy-preserving revocation of VCs. Despite the status of the specification, there are already possibilities to use this revocation method in production. For example, MATTR offers it to its customers on its SSI platform [MA20a], but it can also be implemented independently using open-source libraries such as `vc-revocation-list-2020` [Di21] from Digital Bazaar.

3 Expert Questionnaire

3.1 Preparation

3.2 Questionnaire

3.3 Results

4 Reference Implementation

This chapter focuses on the development of a reference implementation covering the VC lifecycle, which is based on learnings of the last few chapters concerning theoretical background and expert opinions. It is intended to directly address the lack of practical evaluations of SSI solutions in this research area, by leveraging four of the previously listed solutions that can be used to implement the VC lifecycle. In the next sections, the selected solutions will be presented, followed by a description of core principles which form the basis for the development of the reference implementation and its components.

4.1 Provider

This section introduces briefly the solutions selected in section 3.3. From now on, they will be referred to as providers, since they provide the technical basis for the actual implementation.

The first provider is developed by MATTR Limited, which is a New Zealand-based company [MA21h] that specializes in providing solutions for "*[...] a new world of digital trust.*" [MA21d]. This primarily includes their MATTR VII cloud platform, which can be used to technically implement key components of an SSI ecosystem [MA21i]. According to the company, the platform can be divided into the following components: [MA21g]

- *VII Core*: Core includes a variety of REST APIs that serve as the foundation for all interactions. This includes APIs for DIDs, VCs, VPs, and secure messaging between agents using DIDs. [MA21k]
- *VII Extensions*: These are additional components that build on VII Core, such as a bridge for OpenID Connect systems (see user-centric identities), or white label mobile wallets and SDKs. [MA21n]
- *VII Drivers*: Similar to PC drivers, this component allows flexible integration of basic elements. This includes support for various DID methods (did:key, did:web, did:sov, did:ion), crypto suites (ed25519, bls12381g2)[MA21l], and storage options. [MA21m]

In addition, the company offers resources for developers to learn the basics of SSI [MA21j] but also a comprehensive API documentation, tutorials in written or video form [MA21c], mobile wallets, and sample apps [MA21e, MA21k]. The available free tier allows testing the platform, while production systems can be transferred to a usage-based system, where billing is done per transaction [MA21f]. In addition to a variety of DID methods, Mattr offers latest technologies with regard to OIDC, styled (complex) credentials, LD proofs, RevocationList2020, BBS+, and JWM-based messaging as a precursor to DIDComm. Within the SSI community, they participate in the development of open standards [MA21a, LS21b] and software libraries [MA21b] and have demonstrated interoperability in the 2021 Homeland Security Plugfest [Ho21b].

The second used cloud platform provider is developed by the US company Trinsic Technologies Inc. and offers various components for developing SSI solutions [Tr21f]. The company divides the platform into four components:

- *Trinsic Core*: Core offers REST APIs to issue, verify and exchange credentials. [Tr21g]
- *Trinsic Ecosystems*: Enables the creation of ecosystems based on organizations. It can be defined which credentials can be exchanged, who can participate in the ecosystem, and how participants know whom to trust. [Tr21h]
- *Trinsic Studio*: A dashboard that builds a user-friendly GUI on top of the Core APIs to manage organizations, connections, credentials, and verification templates. Trinsic also offers a white-label version of Trinsic Studio. [Tr21i]
- *Identity Wallets*: The wallet SDK allows the creation of cross-platform wallet apps for, for example, Flutter and React Native. Otherwise, Trinsic's own wallet app for Android and iOS can be downloaded from the app stores for testing purposes. [Tr21a]

In addition, Trinsic offers SDKs in various languages that serve as wrappers for the REST API, providing native interfaces for programming languages [Tr21e]. This includes languages such as Ruby, Python, JavaScript and more, with the ability to generate SDKs for additional languages through Trinsic's swagger hub. Worth mentioning is also the well-structured and comprehensive documentation, which also integrates code samples and a getting started tutorial [Tr21b]. The documentation also describes the basics of SSI in a short and concise way, so that even new developers can get an introduction to the topic. The registration process is very short and in combination with the free tier, which allows 50 credentials exchanges per month and has a reduced feature set [Tr21d], it is possible to get started quickly. For production systems, Trinsic charges fixed monthly amounts starting at $18 for 1000 credential exchanges [citation needed]. The company is also partnered with Zapier, which allows custom flows to be created with other apps, e.g. to send a credential via Gmail when there is a new attendee at Eventbrite. Trinsic follows standards like DID, VC, DIDComm and various Aries RFCs, for which open-source work has been done on a .NET implementation [Tr21c]. Its interoperability has not been proven with Homelands Security Plugfest [Ho21b].

Unlike the previous two providers, Veramo is a JavaScript framework which can be used for verifiable data in the SSI context [Ve21c]. It is the direct successor to the uPort project [uP21a], which was started by ConsenSys in 2015 and discontinued in 2021 due to a changing SSI ecosystem and foundational issues. The work on Veramo started around 2020 under the name "DID Agent Framework" and was intended to learn from experiences with uPort by creating a modular architecture whose functionalities could be extended by plugins and used both on the web, mobile, and backend. [uP21b]

Veramo is currently in public beta and is working with the W3C and the DIF [Ve21c]. The center of the framework is the Veramo Agent written in TypeScript, which enables the plugin architecture and exposes its functionality to the developer through a common interface. At all times, the VC and DID standards are respected, while allowing complete freedom and flexibility in all other areas. The Veramo agent implements four basic components for messaging, identifiers, credentials, and keys, which can be seen in figure 4.1. [Ve21d]

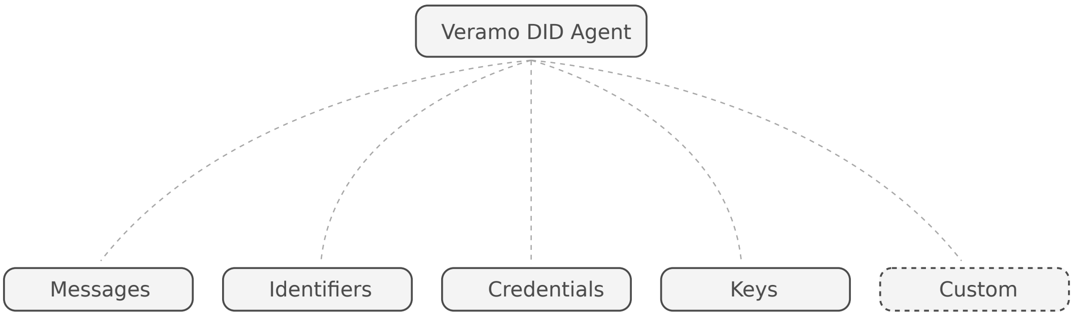


Figure 4.1: Veramo Agent (extracted from [Ve21d])

The framework provides various core plugins, which generally allow things like creating and resolving DIDs based on various DID methods, as well as issuing, retrieving, and exchanging credentials through DIDComm v2 [Ve21d, Ve21a], while providing a template for creating custom plugins [Ve21b]. Similar to MATTR and Trinsic, Veramo provides some posts on fundamentals on VCs and DIDs and some quick guides with sample code on how to use Veramo regarding setup with Node, React, React Native, the CLI, and deployment options of the agent for Heroku and AWS [Ve21d]. Veramo and all its components are completely open-source, with the development team being quite active with commits and answering Q&A as well as tickets on GitHub. A mobile wallet is missing, and its interoperability has not been proven [Ho21b] with Homelands Security Plugfest .

The last provider is again a platform solution: Azure AD for Verifiable Credentials. With this project, Microsoft offers a platform solution to use VCs and DIDs in combination with Active Directory. The project is currently a public preview with which companies can send VCs to users' Microsoft Authenticator apps authenticated through their Active Directory. The app acts as a mobile wallet where users can manage and present their VCs. Microsoft is collaborating with members from the DIF and W3C on this and has followed standards like DID, VC, Sidetree, Well Known DID Configuration, SIOP and Presentation Exchange for their implementation. Its interoperability has not been proven with Homelands Security Plugfest [Ho21b]. [NQ21, Si21, Mi21]

Microsoft uses OpenID Connect to exchange credentials via a token between an Active Directory and the Authenticator app. The credentials are held directly in the app, whose lifecycle can be managed via the Azure AD Verifiable Credential API. This includes APIs for issuance, verification, presentation and more. For the DIDs, Microsoft uses the did:ion DID method, which lets DIDs be anchored on the Bitcoin blockchain via the ION network. The DID document is contained directly in the long form of the DID, or can be retrieved after anchoring on the blockchain via the IPFS network, which is a decentralized file system. ION is a Microsoft-developed layer 2 public permissionless network based on the Bitcoin blockchain. It is based on the DIF specification "Sidetree", which does not need tokens, validators or other consensus mechanisms. Writing and reading payloads on the Bitcoin Blockchain is the only necessity for the network to function. In order for the Microsoft Authenticator app to resolve the DID documents of the ION DIDs, the Microsoft Resolver can be used, which provides an API to communicate with the ION network. The described workings can be seen in figure 4.2. [NQ21]

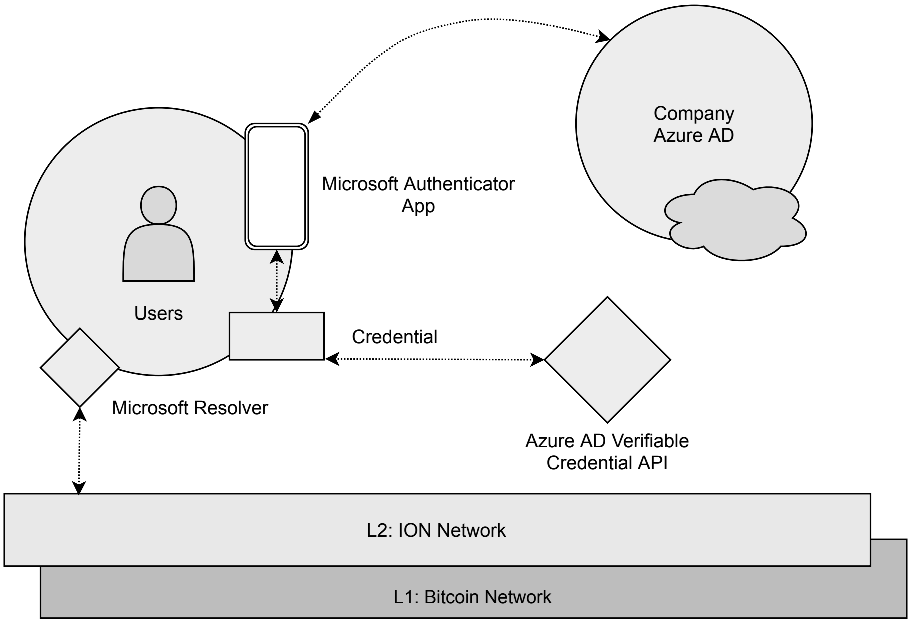


Figure 4.2: Azure AD for Verifiable Credentials (based on [NQ21])

This concludes the description of all four solutions, their structure, and their functionality. In the next section, some core principles and its implications for the reference implementation will be presented.

4.2 Core Principles

As previously stated, the reference implementation should exemplarily implement four providers in such a way that they map, based on their capabilities, the VC lifecycle as much as possible. This enables a practical validation of the promises made by the providers and can thus offer a thorough insight into the existing or missing range of functionality. This way, possible blind spots or even insufficient features can be identified, which can be used to further improve the available solutions. This approach can additionally generate added value for developers who want to use SSI technologies in their projects, since actual experiences, capabilities, and code from a real implementation process can be reused.

Since the results of the implementation are also to be incorporated directly into a new developer-oriented evaluation framework, there are some key considerations that must be defined beforehand. To meet the objectives described in this section, the following considerations were established:

- *Use-case agnostic*: In order to represent the VC lifecycle as broadly and standardized as possible, the reference implementation should not be bound to the requirements and specifics of a use case. Focusing on a specific use-case could possibly lead to certain parts of the VC lifecycle being underrepresented or not implemented at all. Such an open approach can also invite a closer look and implementation of specific facets of a technology that would have been unnecessary for a use case. This also means that the reference implementation must be accessible in such a way that it can be used relatively independently of the technology stack being used for a potential use case.

- *Flexible architecture*: The reference implementation should leverage a software architecture that makes it straightforward to plug in different SSI solutions. Peculiarities and complexities should be abstracted away to create a flexible and resilient architecture.

- *Community efforts*: Since the SSI community is very active, it should be checked beforehand which previous work can be reused for the implementation. This applies to both the architecture and the software libraries used. In this way, it can be ensured that the work does not disregard the considerations and requirements of the community.

- *Implementation experience*: Throughout the entire development process, objective experiences and findings should be documented and summarized. As already mentioned, these may be relevant for other developers, the providers, and also for the mentioned evaluation framework.

Taking these four points into account, the goal is to create a reference implementation that is helpful for developers and meaningful for the evaluation framework. A RESTful API was chosen as the basic implementation form, since it is technology-independent and can be used in various programming languages and environments. It is only necessary that HTTP requests can be sent, which allows a high degree of flexibility in later applications. To implement this, TypeScript was chosen as it was the only language where most solutions offer SDKs for. In addition, other basic libraries in the SSI area are written in TypeScript or at least JavaScript and would thus integrate more easily into the implementation to possibly add missing features. Unlike JavaScript, various features of TypeScript allow cleaner and more robust code [Za20, p. 87] that can simultaneously benefit from much of the existing JavaScript packages being made available from the open-source community. Building on this decision, node.js was selected as the JavaScript runtime that can be used in combination with the express.js library to develop highly scalable web applications such as APIs [Op21a, Op21b]. To make TypeScript work in this environment, various dependencies such as TypeScript itself, ts-node, eslint, and some type definitions were installed via the Node Package Manager. The next section describes how these components and the four solutions were combined into a flexible software architecture.

To enable a flexible software architecture within the described technological skeleton, the factory method and singleton design patterns were chosen, to allow for a pluggable architecture. These considerations resulted in a reference architecture, which can be seen in figure 4.3.

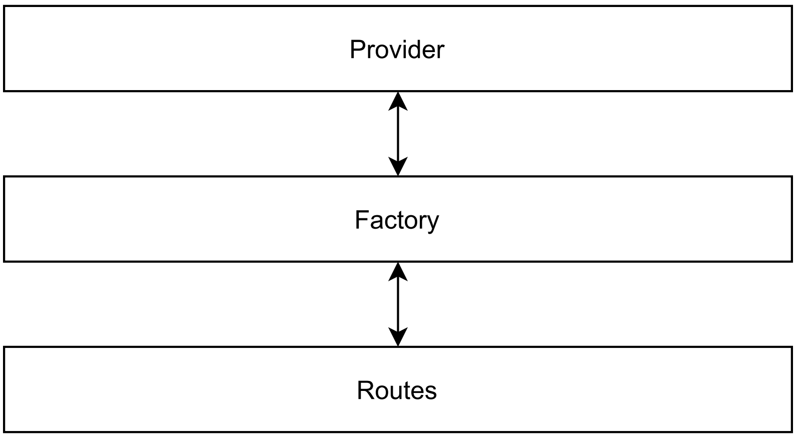


Figure 4.3: Reference architecture

At the center of this architecture is a factory as an abstraction layer, which on the one hand allows plugging in different providers but at the same time mediates between them and the actual routes. Routes are the direct point of contact for requesters of the API and expose the actual functionality to cover the VC lifecycle. More detailed explanations of this layer can be found in the next section and its subsections.

4.3 Routes

As a basis, this work is roughly based on the ideas of the W3C Credentials Community Group, which has created an unofficial draft API definition called “Verifiable Credentials HTTP API”. This describes the structure of an API with all its routes, request, as well as response bodies, which can be used for the VC "lifecycle management". The group defined API contracts according to the OpenAPI standard, which were originally intended to be used as a basis for verifying interoperability of VCs issued by different providers (see interoperability report from [Ho20]). It is important to note that the reference implementation is based on the state of the API contract as of April 2020 (v0.0.2-unstable), as some small things have changed since then. The contract is strongly based on the VC standard and divides the API into resources for the three roles issuer, verifier, and an optional holder. This includes resources for issuing and verifying VCs/ VPs, and deriving, as well as revoking VCs. At this point, it should be emphasized again that this is an API definition and not an actual implementation. [Wo21a, Wo21b]

With regard to the reference implementation, this preliminary work is quite helpful, even if it doesn't fully cover the whole VC lifecycle. Therefore, some changes and additions have been made. A graphical representation of the API definition is shown in figure 4.4, which also allows testing of the individual routes.

All changes made to the original definition are summarized by the following points:

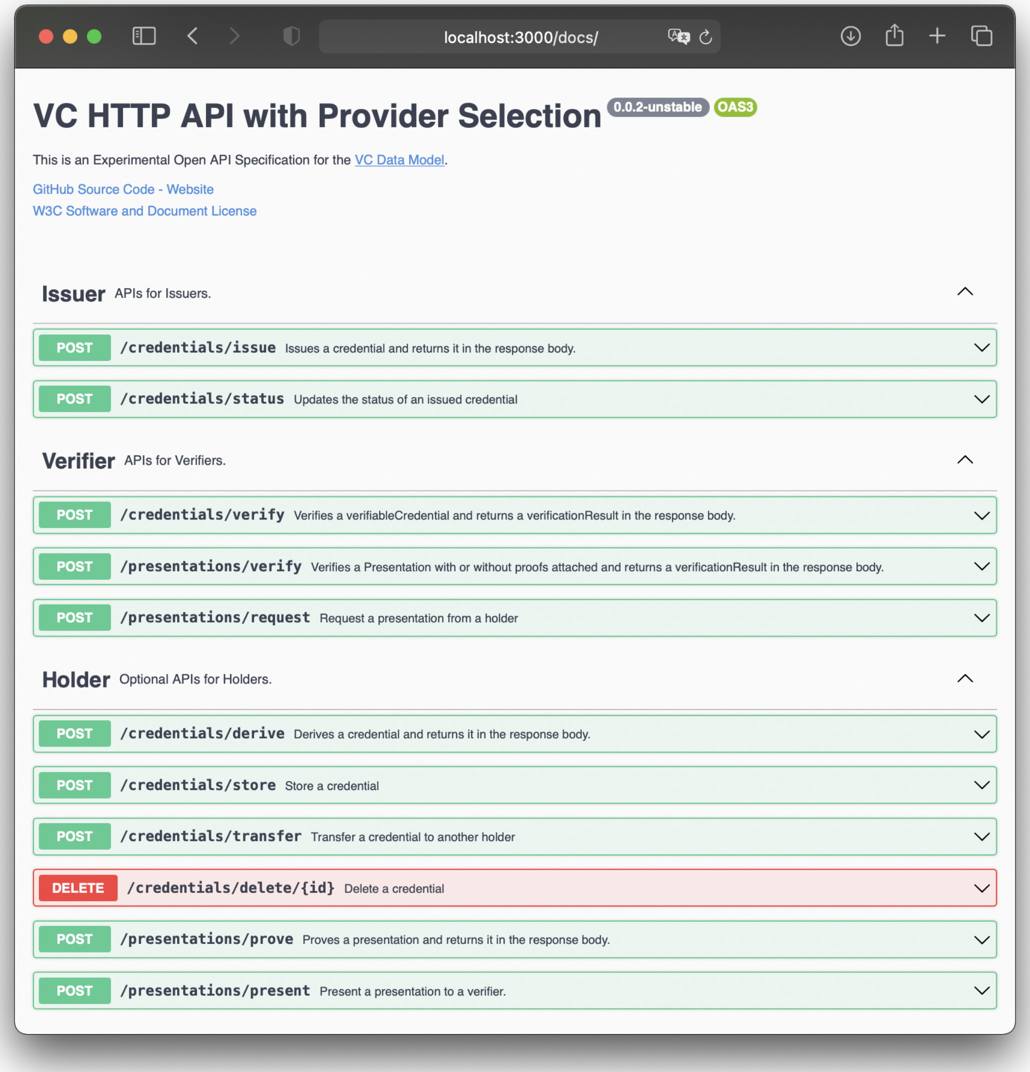
- *Provider selection*: Since four providers should be addressable via the API, a query parameter was added to each route, which is based on a custom provider schema definition. Thus, it is possible to specify which provider should handle a request.

- *Destination selection*: Another query parameter allows defining whether a request targets the local or a remote agent. This mainly includes the issuing route for a VC, which allows a VC to be sent directly to the subject's agent via DIDcomm or QR code. In other cases, such as transferring VCs, presenting or requesting presentations, this can be done directly via the request body. For this purpose, the schema `GenericMessage` was defined, which in its structure is very roughly based on the DIDComm data model with only the most necessary fields.

- *Added routes*: To complete the lifecycle coverage, a route to create a presentation request has been added to the verifier, as well as routes to save, delete, transfer VCs and presenting VPs to the holder. For the request and response bodies, existing schemas were reused as much as possible.

- *Response bodies*: Since some response bodies contained too many unnecessary attributes for the requirements of the API, the schema `GenericResult` was introduced, which only describes whether the operation was successful and whether there were errors. This schema is used for verifying, storing, deleting, and transferring VCs and verifying, as well as requesting/ presenting VPs.

The goal of the customizations was to ensure that the original API definition did not constrain the implementation, while retaining fundamental parts of the community work. Figure 4.4 shows on the one hand all defined routes, but also that they have been divided into the different roles. This concept was also reused for the



localhost:3000/docs/

VC HTTP API with Provider Selection 0.0.2-unstable OAS3

This is an Experimental Open API Specification for the VC Data Model.

GitHub Source Code - Website
W3C Software and Document License

Issuer APIs for Issuers.

- POST `/credentials/issue` Issues a credential and returns it in the response body.
- POST `/credentials/status` Updates the status of an issued credential

Verifier APIs for Verifiers.

- POST `/credentials/verify` Verifies a verifiableCredential and returns a verificationResult in the response body.
- POST `/presentations/verify` Verifies a Presentation with or without proofs attached and returns a verificationResult in the response body.
- POST `/presentations/request` Request a presentation from a holder

Holder Optional APIs for Holders.

- POST `/credentials/derive` Derives a credential and returns it in the response body.
- POST `/credentials/store` Store a credential
- POST `/credentials/transfer` Transfer a credential to another holder
- DELETE `/credentials/delete/{id}` Delete a credential
- POST `/presentations/prove` Proves a presentation and returns it in the response body.
- POST `/presentations/present` Present a presentation to a verifier.

Figure 4.4: Modified API definition (based on [Wo21a])

implementation by dividing the routes into `HolderRoutes.ts`, `VerifierRoutes.ts`, and `IssuerRoutes.ts`. These implement the code for the routes they contain by accessing the provider classes generated by the factory. Further descriptions of the design patterns built on top of these are discussed in more detail in the next section.

4.4 Factory

As already mentioned in section 4.2, the factory method and singleton design patterns were chosen for the software architecture. Factory method belongs to the creational patterns and thus influences how the instantiation process is carried out. A developer can thereby decide independently of the system how objects are created, which enables a high degree of flexibility. It defines an interface or an abstract class for the creation of objects, whereby the instantiation of objects is done by subclasses instead of a class. This is useful, for example, if a class does not yet know which objects it needs to create at runtime. [Ga95, pp. 81, 85, 107-108]

This pattern is appropriate because a request determines which solution and thus which objects have to be created. In addition, it allows the complexities of the individual solutions to be abstracted away, so that when defining the individual routes, only the concrete factory class must be called, which returns the correct object of the requested solution. This way, the routes only need to be programmed once and additional solutions can be added afterwards without having to change the code of the routes. Figure 4.5 shows a UML diagram that represents the concrete factory method pattern in the reference implementation. The interface `Factory` defines a method `createProvider()`, which is implemented by the class `ServiceProviderFactory`. This is the class which is instantiated, for example, in the routes and which is used to retrieve the object of a provider. A provider is the class of one of the four solutions that implements basic methods like the VC issuance defined by the `ServiceProvider` interface.

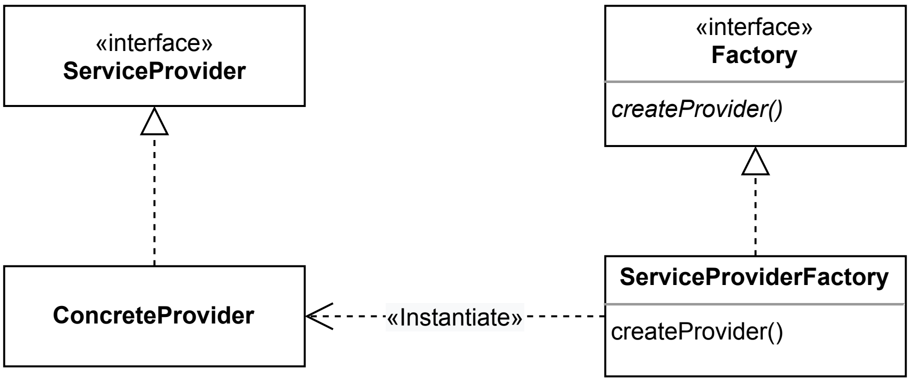


Figure 4.5: Factory method pattern in reference implementation (extracted and edited from [Ga95, p. 107])

As opposed to this, the singleton pattern is used in the individual concrete provider classes and allows that only one globally callable instance of a provider class can be created [Ga95, p. 127]. The rationale for this is that no more than one object is needed, caching is simplified, and multiple provider objects could lead to unforeseen complications, especially in messaging flows.

How these design patterns were implemented in the reference implementation can be seen as an example in the following listings. The starting point for the example is a route for verifying a VC in the `VerifierRoutes.ts`, seen in listing 4.1.

```
const router = express.Router();
const factory = new ServiceProviderFactory();
router
  .post("/credentials/verify", providerCheck, async(req, res) => {
```

```
5      const query = req.query.provider.toUpperCase();
6      const body = req.body.verifiableCredential;
7      const provider = factory.createProvider(ServiceType[query]);
8      const result: GenericResult =
9        await provider.verifyVerifiableCredential(body);
10
11     if (result instanceof Error) {
12       res.status(500).send(<GenericResult>{
13         success: false,
14         error: result.message
15       });
16     } else {
17       res.status(200).send(result);
18     }
19   })
20   ...
21 export = router;
```

Listing 4.1: Extract of verifier routes

Here, the POST method for verifying a VC is attached to the `router` (line 4). A `providerCheck` is appended as middleware, which checks whether the provider specified in the query is indeed valid. Within the route body, the provider object is created via the `ServiceProviderFactory`, which should handle the request (line 7). This object is then used in lines 9 to verify the VC in the request body, the result of which is then sent as a response to the requester. By exporting the router object (line 21), all injected routes can be imported into `index.ts` and made available. This example shows that no provider-specific code is present in the route code due to the factory method pattern. To demonstrate how the `ServiceProviderFactory` works, its code can be seen in listing 4.2.

```
export class ServiceProviderFactory implements Factory {
  createProvider(type: ServiceType): ServiceProvider {
    switch (type) {
      case ServiceType.VERAMO:
        return VeramoProvider.getInstance();
      case ServiceType.MATTR:
        return MattrProvider.getInstance();
      case ServiceType.TRINSIC:
        return TrinsicProvider.getInstance();
      case ServiceType.AZURE:
        return AzureProvider.getInstance();
      default:
        return null;
    }
  }
}
```

Listing 4.2: Extract of service provider factory

The class ServiceProviderFactory implements the createProvider method according to the Factory interface. If this method is called with the desired provider (ServiceType), as e.g. in listing 4.1 line 7, the singleton object corresponding to the provider is returned via a switch statement. According to the factory method pattern, all provider classes implement the ServiceProvider interface and its signatures as concrete methods. This can be seen exemplarily in listing 4.3.

```
export interface ServiceProvider {
  deleteVerifiableCredential(identifier: string):
   Promise<CredentialDeleteResult>;
  ...
}

export class VeramoProvider implements ServiceProvider {
  async deleteVerifiableCredential(identifier: string):
   Promise<CredentialDeleteResult> {
    const db = new VeramoDatabase();
    const result: CredentialDeleteResult = { isDeleted: false };
    try {
      const isDeleted = await db.deleteCredential(identifier);
      result.isDeleted = isDeleted[0];
      result.message = isDeleted[1];
      return result;
    } catch (error) {
      return error;
    }
  }
  ...
}
```

Listing 4.3: Example of provider implementation

In this case, the class VeramoProvider implements the interface ServiceProvider with its signatures like deleteVerifiableCredential() concretely, to delete a VC from the Veramo agent database.

Now that the factory layer itself and how it acts as a mediator between the other layers has been described, the next section will focus on the concrete integration of the selected providers and its resulting architecture.

4.5 Provider Integration

Having looked at the general architecture and its concrete implementations concerning routes and abstractions, this section focuses on the integration of the individual providers. For this purpose, figure 4.6 shows the concrete architecture of the reference implementation, which is based on the previously defined reference architecture.

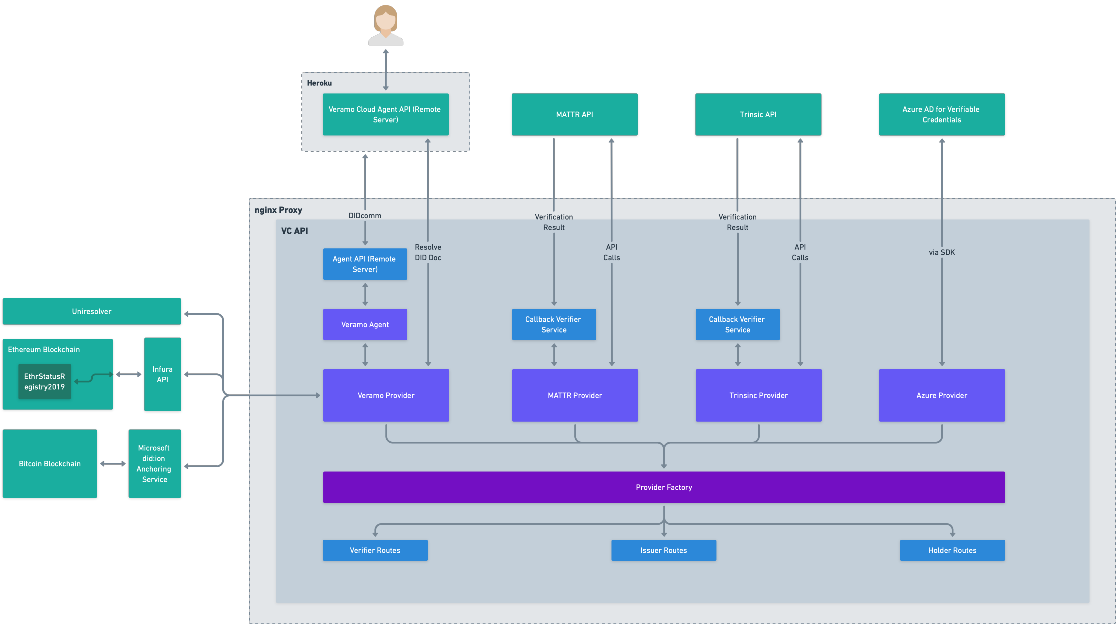


Figure 4.6: System architecture (TODO: Vector redo)

This contains the previously defined types of routes at the bottom, but also the provider factory and the four providers above it. Added at this point are mainly provider-specific elements like callback services, but also an nginx proxy that surrounds the reference implementation. This proxy is assigned a domain so that external components can correctly address internal components such as callback or messaging services. The specific provider implementation, its peculiarities and gained experience will be discussed in more detail in the next subsections for each provider.

4.5.1 Mattr

With regard to the integration of the Mattr provider into the reference implementation, the generous free tier and the well-structured documentation proved to be very helpful. Thus, a large part of the VC lifecycle could be integrated relatively unproblematically by simply addressing the corresponding endpoints of the MATTR VII REST API. These take care of any logic and can also be used to manage DIDs and their keys as well as VCs. The support of BBS+ for ZKP credentials, revocable credentials (RevocationList2020) and LD proofs is directly integrated into the API as well. For mobile wallet interactions concerning VC issuance, an OIDC provider (see user-centric identity) like Auth0 was necessary at the time of implementation. Its initial setup took some time, but was feasible due to the good documentation. By now, VCs can also be issued directly using Mattr's JWM module, QR code, or deep link to the Mattr mobile wallet without an OIDC provider. With regard to presentation requests and its verification results, the sample apps were used to integrate a callback service into the reference implementation, although this flow can also be implemented via OIDC with greater effort. The Mattr team also seems to actively maintain their external Slack channel, as they are relatively quick to respond to questions and issues from platform users there, and even offer private calls in some cases. In summary, the broad coverage of the VC lifecycle, the large number of features, the developer-friendly and detailed documentation with tutorials stood out positively. Mattr offers developers deep but simple and well-documented access to various current technologies and facets of SSI.

Nevertheless, some things were noticed that could be problematic and restrictive under certain circumstances. For example, on-boarding to the platform has been relatively cumbersome in April 2021, as it was necessary to join the official Slack channel after successful registration, where support then triggers the rest of the process manually. After the cloud agent had been created, the password was communicated by the support via Keybase. An automated process without the need for Slack and Keybase would certainly have been much more pleasant, whereas this seemed to be only an interim solution. Additionally, some features are not or only partially usable in combination, such as the support of BBS+ only for credentials based on did:key or did:ion. Finally, it should be noted that the private keys for all DIDs are managed by MATTR and cannot be exported by users. This reflects the platform nature of Mattr, which does not allow developers to extend basic functionality. If the existing features are sufficient, this is certainly not a problem, but the resulting inflexibility can be restrictive and make custom flows impossible. So potential developers will need to look closely at whether Mattr's approach aligns with their goals and product strategy.

The majority of the integration took place in `MattrProvider.ts`, which implements the `ServiceProvider` interface. Since this class implements any functionality via HTTP requests (see figure X), listing 4.4 is an example of how this works. However, it should be noted that methods such as `issueVerifiableCredential()` are significantly more complex, as specific logic such as the distinction in issuance to a wallet or not must be considered accordingly.

```
async verifyVerifiablePresentation(body: VerifiablePresentation):
 Promise<GenericResult> {
  const request = { presentation: body };
  const authToken: string = await this.getBearerToken();
  const result: GenericResult = {
   success: null,
  };

  try {
    const response = await fetch(`.../presentations/verify`, {
      method: "POST",
      body: JSON.stringify(request),
      headers: {
        "Content—Type": "application/json",
        Authorization: `Bearer ${authToken}`
      },
    });
    const verificationResult = await response.json();
    result.success = verificationResult.verified;
    result.error = verificationResult.reason;
    return result;
  } catch (error) {
    return error;
  }
}
```

Listing 4.4: Example of mattr verification implementation

Furthermore, helper methods were implemented to generate authentication tokens for the requests or to cache QR codes for issuing VCs to mobile wallets. Especially, interactions with the latter required several extra implementations that enable the generation of issuance and presentation requests in the form of QR codes via an OIDC provider to MATTR. Starting with the issuance of a VC to such a wallet, the type, and attributes of the VC must be prepared at the OIDC provider and MATTR. The resulting provider ID can then simply be referenced in an issuance URL and, if desired, encoded in a QR code. By scanning this, the associated VC can be obtained directly in the MATTR wallet app. This code is part of the `MattrVerifierService.ts` and can be seen in listing 4.5.

```
private getOIDCIssuerQRCode(oidcIssuer: string): Buffer {
  if (this.issuerQrCache.has(oidcIssuer))
    return this.issuerQrCache.get(oidcIssuer).image;

  const qrcode: Buffer = qr.imageSync(
    `openid://discovery?issuer=${process.env.MATTR_URL}
    /ext/oidc/v1/issuers/${oidcIssuer}`,
    { type: "png" }
    );
  this.issuerQrCache.set(oidcIssuer, qrcode);
  return qrcode;
  }
}
```

Listing 4.5: OIDC issuance QR code generation

To verify a VC from the MATTR wallet app, a presentation request must be prepared. For this purpose, a presentation template is first defined on the MATTR platform, which contains, for example, the allowed issuers or the requested attributes of a requested VC. MATTR assigns a unique ID to this template. This ID can then be used in the first step of provisioning in the reference implementation. Here the actual presentation request is prepared with the verifier DID, the template ID, an expiration date and a callback URL via the MATTR API. In the next step, authentication keys are retrieved as a DID URL from the verifier DID document via the MATTR platform to sign the presentation request via the same platform in the next step. The resulting JWS payload is cached and a QR code is generated with a public URL of the reference implementation. This process can be seen in listing 4.6.

```
public async generateQRCode(request: GenericMessage): Promise<Buffer> {
  const templateId: string = request.body.request.credentialType;

  // Check cache
  if (this.qrCache.has(templateId))
    return this.qrCache.get(templateId).image;

  // Prepare QR code and JWS payload URL
  const publicUrl = this.publicUrl;
  const provisionRequest =
    await this.provisionPresentationRequest(publicUrl, request);
  const didUrl = await this.getVerifierDIDUrl(request.from)
  const didcommUrl =
    await this.signPayload(publicUrl, didUrl, provisionRequest);
  const qrcode = qr.imageSync(didcommUrl, { type: "png" });

  // Cache and return it
  this.qrCache.set(templateId, qrcode, request.expiresTime);
  return qrcode;
}
```

Listing 4.6: Generate QR code for OIDC presentation reqest

When the QR code is scanned with the MATTR wallet app, the qr route of the `MattrVerifierRoutes.ts` is called, which returns the stored JWS payload url. This is the starting point for all further interactions between the wallet app and the MATTR platform. The callback route, see callback verifier service in figure X, in the reference implementation allows MATTR to report whether the presentation was successful and the VC could be verified. Both routes were defined via the express router.

This describes the fundamental aspects of the MATTR implementation. In the next subsection, Trinsic will be discussed in the same context.

4.5.2 Trinsic

The integration of Trinsic into the reference implementation proved to be pleasant. This was mainly due to the documentation but also due to their Trinsic Studio. With the latter, an organization and associated credential and verification templates could be created within a few minutes using a simple dashboard. This is however also possible programmatically via their API, which leaves room for automation. The resulting IDs could later be used to trigger the issuance and presentation flows using

Trinsic's JavaScript SDK, which also includes type definitions for TypeScript. No other solution made the implementation process so quick and easy.

The Hyperledger focus is, however, noticeable from a technological point of view. Only JSON credentials and CL signatures for revocation, DIDComm v1, Aries exchange protocols and Hyperledger-specific DID methods like did:sov are supported. Newer technologies as well as DID methods based on public permissionless blockchains would be appreciated here. In addition, functionality is abstracted away even further in contrast to MATTR, which increases user-friendliness but hardly allows any flexibility. Thus, no custom DIDs can be generated, and no VCs in can be directly issued or verified without first generating a template. Direct access for generating VPs on the cloud agent or the selective disclosure functions is also not available. All these things are managed in the background by Trinsic and the mobile wallet. This can certainly be sufficient for certain use cases, but maybe too inflexible for others. Just like MATTR, developers are completely dependent on Trinsic in terms of the tech stack and private key management. The company is trying to address many of the points mentioned above with the Core v2 platform, which is currently in closed beta. Latest technologies such as JSON-LD credentials, LD proofs, BBS+, presentation exchange, DIDComm v2 and new DID methods such as did:key or did:web are supported here. Trinsic could thus catch up with MATTR in terms of supported technologies. Table 4.1 shows the current roadmap of the Trinsic platform.

Table 4.1: Trinsic Roadmap (based on [Ri21b])

Type	Current	Beta	Future
Data Exchange	JSON	JSON-LD	JWT
	CL signatures	BBS+ signatures	OIDC SIOP
	Aries exchange	Presentation exchange	
Communication	did:peer	did:key	WACI
	DIDComm v1	DIDComm v2	BLE, NFC
	HTTP transport	gRPC transport	
Public Trust	did:sov	did:key, did:web	did:un, did:ion
	Hyperledger Indy	did:indy	

The main part of the implementation takes place in the `TrinsicProvider.ts`, which implements the `ServiceProvider` interface. To communicate with the Trinsic API, the JavaScript SDK was used to create an object of the `CredentialsServiceClient` class. This is done in the constructor so that the object is available immediately after initialization. This can be seen in listing 4.7.

```
export class TrinsicProvider implements ServiceProvider {
  client: CredentialsServiceClient;

  private constructor() {
    this.client = new CredentialsServiceClient(
      new Credentials(process.env.TRINSIC_KEY),
      { noRetryPolicy: true }
    );
  }
  ...
}
```

Listing 4.7: Connecting to Trinsic API via SDK

How to programmatically issue a VC to the mobile wallet via Trinsic, for example, can be seen in listing 4.8. As required by the `ServiceProvider` interface, this is done by implementing the `issueVerifiableCredential()` method that first checks whether the requester actually wants to issue a pre-defined credential to the wallet. If this is not the case, it is informed that only predefined credentials can be issued to a wallet. If everything is correctly defined, the ID of the credential template as well as the values for the attributes of the credential are obtained from the request body. The `GenericMessage` schema in the request body is used for this. An object is formed from these values, which is submitted to the API via the Trinsic SDK and results in a `CreateCredentialResponse` object. The `offerUrl` contained therein is then encoded in a QR code, which can then be scanned via the Trinsic wallet to retrieve the VC.

```
async issueVerifiableCredential(
  body: IssueCredentialRequest | GenericMessage,
  toWallet: boolean
): Promise<IssueCredentialResponse | Buffer> {
  try {
    if (!toWallet) throw Error("Only issuance to Trinsic wallet...");
    if (isGenericMessage(body)) {

      // Prepare request body for Trinsic
      const message: GenericMessage = body;
      const request = {
        definitionId: message.body.credentialType,
          connectionId: null,
          automaticIssuance: false,
          credentialValues: message.body.claimValues,
      };

      // Generate QR code with offer URL
      const vcOffer: CreateCredentialResponse =
        await this.client.createCredential(request);
      const qrcode: Buffer =
        qr.imageSync(vcOffer.offerUrl, { type: "png" });

      return qrcode;
    } else {
      throw Error("Issuing manual VCs is not supported...");
    }
  } catch (error) {
    return error;
  }
}
```

Listing 4.8: VC issuance with Trinsic

In the case of Trinsic, 3 of 10 methods of the `ServiceProvider` interface could be implemented directly, while all parts except for the transfer of VCs are at least indirectly part of Trinsic. Among them are the methods for issuing, deleting and creating presentation requests. The method for revoking credentials could have been implemented theoretically, but was not part of the free tier. After a presentation request has been successfully performed by the user, the result of the verification can be received via a webhook in the reference implementation. Similar to MATTR, a route was created via Express, to which the Trinsic platform can send the result. This webhook URL also had to be added via the Trinsic Studio beforehand, since the URL is not part of the presentation request body, as opposed to MATTR. Listing 4.9 shows how the webhook route is implemented. This forms, together with the verification method, the callback verifier service as shown in the system architecture in figure 4.9.

```
const router = express.Router();
const trinsic = TrinsicProvider.getInstance();

router.post("/webhook", async (req, res) => {
  try {
    if (req.body.message_type === "verification") {
      const verification =
        await trinsic.client.getVerification(req.body.object_id);

      console.log(verification);
    }
  } catch (error) {
    console.log(error.message || error.toString());
  }
  res.status(200).end();
});

export = router;
```

Listing 4.9: Trinsic webhook for verification result

At this point, all the specifics of the Trinsic platform and its integration into the reference implementation have been addressed. In the next subsection, the first and only non-platform solution and its implementation will be presented.

4.5.3 Veramo

The integration process of Veramo into the reference implementation proved to be a mixed experience. Especially the CLI offered a good first start to try out Veramo. Thus, all basic features like creating DIDs and issuing and verifying VCs could be tested in advance without any implementation process. Within the implementation, the very high degree of flexibility was especially noticeable. None of the previous solutions supports so many DID methods (did:key, did:web, did:ethr, did:ion), whereas the connection to the Uniresolver allows resolving almost any DID. In terms of deployment options, the Veramo agent can be used in web frontends, smartphone apps, and backend systems through the TypeScript backend, while the agent's modularity also allows tasks to be distributed among different agents. This allows for different architectures where, for example, one Veramo cloud agent implements only key management, another implements the issuing of VCs, and the agent in the mobile wallet implements only messaging and credentials storage. Since the Veramo agent also exposes all of its functionalities via a REST API with OpenAPI documentation, similar to MATTR and Trinsic, all functions can also be used on non-supported platforms via HTTP requests. This is not possible with any of the other solutions. In addition, with the DIDComm v1 implementation, complex communications between Veramo agents could also be implemented, such as the transfer of credentials between holders. Because of the open access to these modules, pretty much any use case can be realized, which is not possible due to limitations in the other solutions. Since the integration, Veramo released a DIDComm v2 module which replaces the old alpha version of DIDComm v1 [Ve21a]. Especially for such messaging cases, it was very helpful that a Veramo agent could be created on Heroku with a one-click button, making such multi-agent flows testable. In addition, the React Native integration was tested, with which a demo mobile wallet could be created within a very short time, with which a user could manage and receive VCs. In contrast to the platform solutions, the flexibility but also the complete control over architecture, code, and data such as the VCs and the private keys of DIDs stand out.

Nevertheless, there were a few things that complicated the implementation process. The now updated documentation was either outdated or not available at all in April 2021. Only the code of the CLI implementation could be used as a guide, which in combination with some guesswork and trial and error led to the desired goal. Even today, not all areas and features are covered so far, such as code and explanations for the messaging system, how to issue and verify credentials, how to resolve DIDs, that there is a did:ion plugin, types of events in the event system and much more. The documentation really only covers the most necessary areas to get started, which may discourage some developers. Furthermore, there are some technological limitations, which are briefly listed below:

- *Credential type*: Currently only JWT credentials are supported. Support for JSON-LD signatures is currently being worked on [github citation needed].

- *Revocation*: There are no official, up-to-date plugins for revocation. There is only a library from two years ago that can be used with manual effort in Veramo to create revocable credentials with did:ethr and later revoke them as well. Support for RevocationList2020 does not exist, but can be retrofitted with open libraries from digital bazaar due to Veramo's open architecture.
- *Selective Disclosures & ZKP*: Due to the nature of JWT credentials, this is not currently possible. Veramo recommends keeping credentials as atomic as possible so that the user can present individual credentials with only a few attributes. Additionally, Veramo currently uses a proprietary exchange data format instead of the DIF Presentation Exchange format. However, support for BBS+ in combination with LD proofs and Presentation Exchange is planned [TCSL21].
- *Verification*: There is no proper API for verifying VCs and VPs. Meanwhile, only the validity of the JWT can be checked, but it does not retrieve the DID document for the issuer's public key, does not check for revocation, integrity, or other important parameters. The developer currently has to check all of these by itself. However, this is also being looked at by the development team, which is working on a comprehensive verification API [Ri21a].

This makes it clear that Veramo really is a beta and that various areas are not yet ready. Nevertheless, it should be noted that, in contrast to the other solutions, the open architecture means that missing functions can be retrofitted at any time. This behaviour is also supported by the Veramo developers, as they want an ecosystem of community plugins.

Looking at the reference implementation, there is more effort here compared to the platform solutions, as now any logic has to be implemented via the Veramo API itself. In the middle of this is the `VeramoProvider.ts`, which implements the `ServiceProvider` interface and thus all lifecycle-specific methods. In order to access the methods of the Veramo API and its included plugins, an object of the Veramo agent is created and exported in the `VeramoSetup.ts` so that it can be directly imported and used in various places. To create the agent, all necessary plugins must be imported in the form of libraries and taken into account when initializing the object. This can be seen exemplary in listing 4.10.

```
// Core interfaces
import { createAgent, IDIDManager, ... } from "@veramo/core";

// Core identity manager plugin
import { DIDManager } from "@veramo/did-manager";

// Credential Issuer
import { CredentialIssuer, ICredentialIssuer } from "@veramo/credential-w3c";

...
```

```
export const veramoAgent = createAgent<
  IDIDManager & IKeyManager & IDataStore & IResolver & ... >({
  plugins: [
    ...
    new KeyManager({
      store: new KeyStore(dbConnection, new SecretBox(secretKey)),
      kms: {
        local: new KeyManagementSystem(),
      },
    }),
    new DIDManager({
      store: new DIDStore(dbConnection),
      defaultProvider: "did:key",
      providers: {
        "did:key": new KeyDIDProvider({
          defaultKms: "local",
        }),
        ...
      },
    }),
    new DIDResolverPlugin({
      resolver: new Resolver({
        ...
        key: getDidKeyResolver().key,
        ...getUniversalResolverFor(["io", "elem", "sov"]),
      }),
    }),
    new CredentialIssuer(),
    new MessageHandler({ ... }),
    ...
  ],
});
```

Listing 4.10: Veramo agent creation

This is only a small excerpt, but it demonstrates the rough concept and functionality. For the reference implementation, various plugins were implemented for storing and managing keys, DIDs, VCs, VPs as well as messages, message handlers (DIDComm, JWT, ...), DID provider, DID resolver and credential issuance. In addition, event listeners were utilized to document verification results and submitted presentation requests. Especially the latter was helpful in testing the multi-agent messaging flows to let a cloud agent automatically respond to submitted presentation requests via its API. This part is relatively well documented, but there were problems implementing the message handlers correctly, which resulted in errors with the validation of messages like JWT credentials or DIDComm messages. The author's issue report on GitHub was answered by the Veramo developers on the same day. It was described that the order in which the different message handlers are integrated in the setup is significant and was incorrect in the reference implementation. Such small details are unfortunately not documented and cost considerable time to debug. That being said, the author was able to identify an actual bug during implementation where mismatches of signatures between a VC and its VP occur under certain Partialumstances. This report was also being responded to on the same day and a fix was rolled out in under two weeks. The developers appear to respond quickly and helpfully in general on GitHub [github issues citation].

With regard to the actual implementation of the lifecycle in the `VeramoProvider` class, the methods of the Veramo agent proved to be well usable. Basic methods for creating VCs, VPs, verifying messages like JWT credentials and sending presentation requests are provided, the latter being called Selective Disclosure Requests by Veramo. Looking at the descriptions of this topic in subsection 2.5.2 and the nature of JWT credentials, the term selective disclosure is not well-chosen, since only a set of credentials and not a set of attributes of one credential are disclosed here. In listing 4.11 is a code snippet for creating a VC using the Veramo agent.

```
async issueVerifiableCredential(body: IssueCredentialRequest,
 toWallet: boolean): Promise<IssueCredentialResponse> {
    try {
      body.credential.issuer = { id: body.credential.issuer.toString() };
      const save: boolean = body.options.save ? body.options.save : false;
      const credential: W3CCredential = body.credential;

      const verifiableCredential: W3CCredential =
       await veramoAgent.createVerifiableCredential({
        save: false,
        credential,
        proofFormat: "jwt",
      });

      // Prepare response
      const result: IssueCredentialResponse = {
        credential: verifiableCredential,
      };

      if (toWallet) {
        try {  // Send VC to another Veramo agent
          const msg = await veramoAgent.sendMessageDIDCommAlpha1({
            save: true,
            data: {
              from: verifiableCredential.issuer.id,
              to: verifiableCredential.credentialSubject.id,
              type: "jwt",
              body: verifiableCredential.proof.jwt,
            },
          });
31          result.sent = true;
32          return result;
33        } catch (error) {
34          return error;
35        }
36      }
37      return result;
38    } catch (error) {
39      return error;
40    }
41  };
```

Listing 4.11: Issue a VC with Veramo

At the beginning, the credential object is prepared, which is then converted to an VC in lines 8 to 13 using the methods After that, the API response is prepared with the created VC, or if an error occurs during issuance, the error is sent directly to the requester as a response. If the request to the API defined that the VC should be sent directly to the agent of the DID via the messaging endpoint, lines 21 to 35 handle this, using the `sendMessageDIDCommAlpha1()` method. This method has been deprecated and should be replaced by the new DIDComm v2 implementation. Considering the scope of this work, a short-term implementation of the new module has been omitted at this point. This feature is useful, for example, to send a VC directly to the Veramo agent of a mobile wallet. MATTR and Trinsic offer similar functions as an alternative to scanning QR codes.

In addition, four other files and classes were created to implement or retrofit Veramo-specific features. These are briefly described below:

- *VeramoDatabase.ts*: At the time of development, there was no method to delete VCs from the local flatfile database. Since the database is in the sqlite format, the `sqlite3` library was used to add that functionality. Since v2.1.0 this is also a native DataStore functionality.

- *VeramoRevoker.ts*: This class implements uPort's `ethr-status-registry` library and retrofits the functionality to revoke an VC with a did:ethr within an on chain smart contract on the Ethereum Blockchain. This requires a connection to an Ethereum node or a service provider such as Infura.

- *VeramoRemoteAgent.ts*: This class can connect to another Veramo agent that one has control over. This is more of a helper class that can be used for testing purposes, for example, to force a cloud agent to automatically respond to a presentation request via its RESTful API.

- *VeramoAgentAPI.ts*: Here, the local Veramo agent exposes its methods via a RESTful API and the associated API docs. Furthermore, a did:web is automatically set up for the external URL and the associated messaging endpoint. This is relevant for multi-agent communication flows.

Flexibility and liberty are the main advantages of Veramo, yet the beta status was noticeable during the implementation. Some central features are not or only partially available, and from the above description it is clear that a lot is happening. Within a few months, some features were added that made some implementations (see VeramoDatabase.ts, DIDComm) obsolete. In the next subsection, Azure AD for Verifiable Credentials will be discussed as the last of the four solutions.

4.5.4 Azure

With regard to the implementation, in contrast to MATTR and Trinsic, Active Directory still had to be prepared as the platform. Since the extension for Verifiable Credentials is still in public preview, a P2 licence is required, which is associated with higher costs. Alternatively, a developer account can be requested from Microsoft. Within the client a new resource group is created, which must contain a key vault for keys, a storage account for credential specific rules and display JSON files and various other settings. Microsoft provides detailed documentation with descriptions and illustrations for these steps, so this process was fairly straightforward. Nevertheless, this can definitely be confusing in some places for Azure beginners. In the next step, the actual Verifiable Credential can be defined in its content and representation, which is also described in the documentation. Azure automatically assigns a URL to this VC schema, which means that it can be retrieved via an OpenID provider. There are other steps, such as creating an application in Azure so that issuance and verification can take place in the reference implementation. So positive was the good and clear documentation to prepare a client to issue and verify VCs. Furthermore, it is certainly interesting for companies that already use Microsoft products and Active Directory. The VC solution could be integrated into existing architectures without much effort. Moreover, it is one of the few solutions that uses ION as the DID method, relying on a public permissionless blockchain, thus actually creating independent DIDs that can also embed metadata in their DID documents. With Azure, VC can be issued to the mobile app in JWT format and also presented from there as a VP on the basis of a corresponding request. In addition, VCs can be revoked through the Azure portal.

Nevertheless, there are some points that make the Microsoft solution appear incomplete, and lack transparency compared to the other solutions. For example, Azure abstracts various logic and complexities similar to Trinsic, with the difference that even fewer features are available, and it is even less clear which standards are used at which point. Through debugging sessions of the author, it was possible to determine that simple JWT credentials are exchanged and propriety revoking mechanisms are used. Moreover, unlike MATTR and Veramo, VCs cannot be issued and verified directly based on a JSON-LD input either, which means that only VCs whose schema has been defined in Azure beforehand can be issued or verified. In addition, there appears to be no official API for revoking credentials yet, which is why it was not possible to implement this in the reference implementation. Furthermore, routes for storing, verifying, and presenting without request could not be implemented because no APIs are offered for this. These steps are abstracted away in other process steps, while the transfer and derivation for ZKPs of VC is not possible at all. Due to the missing functionalities, it is impossible to implement the complete lifecycle with its individual steps. In addition, the example projects with code for issuers and verifiers are functional but not documented at all, which led to some confusion. Furthermore, basic type definitions for TypeScript seem to be flawed and even so the code seems to be questionable. There was a GitHub issue [Ye21] about this, which the author of this work also responded to, and was eventually closed without comment by the repository owner. The original creator of the issue announced that they ultimately decided against the Azure solution. With correspondingly messy modifications, functionalities for issuance and presentation request routes could still be implemented. Additional DID methods, revocations standards, ZKP technologies such as BBS+ and associated LD proofs have not been announced. In terms of the basic approach, Microsoft's solution is most similar to Trinsic, but is not on a par in terms of functionality, documentation, SDKs and associated examples.

The main part of the implementation takes place in `AzureProvider.ts`, which implements the `ServiceProvider` interface. Unique aspects are that at the beginning of the class, an object of the `CryptoBuilder` class of the verifiablecredentials-verification-sdk-typescript library and a request cache are created as `Map<string, any>`. The former can be used to create the requests for Azure, which are then cached in the request cache, which becomes relevant in later process steps. Listing 4.12 shows an example of how to create an issuance request via the Azure SDK.

```
async issueVerifiableCredential(body: GenericMessage,
 toWallet: boolean): Promise<Buffer> {
    try {
      if (!toWallet) throw new Error("Only issuance to wallet...");
      if (!(body.from && body.body.request.credentialType))
       throw new Error("Please define from and credentialType");

      const requestBuilder = new RequestorBuilder(
        {
          ...
          presentationDefinition: {
            input_descriptors: [
              {
                id: "credential",
                schema: {
                  uri: [body.body.request.credentialType],
                },
                issuance: [
                  {
                    manifest: body.from,
                  },
22                  ],
23                },
24              ],
25            },
26          } as IRequestor,
27          this.crypto
28        ).allowIssuance();
29
30        const issuanceRequest = await requestBuilder.build().create();
31        const sessionId = uuidv4();
32        this.requestCache.set(sessionId, issuanceRequest.request);
33
34        const requestUri =
35         encodeURIComponent(`/azure/issue-request.jwt?id=${sessionId}`);
36        const issueRequestReference =
37         "openid://vc/?request_uri=" + requestUri;
38        const qrcode: Buffer =
39         qr.imageSync(issueRequestReference, { type: "png" });
40        return qrcode;
41      } catch (error) {
42        return error;
43      }
44  }
```

Listing 4.12: Create a VC issuance request with Azure

In the beginning, some error handling is done, which can return corresponding errors to the requester. If both the credentials type and its schema URL from Azure are included in the request body, an issuance request object is created between lines 8 and 30. This is cached in the request cache with a corresponding UUID (line 32). This UUID is integrated into an issuance URL (line 34 – 37), which leads to a route within the reference implementation. In order for this to be accessed by Microsoft Authenticator, the URL is encoded into a QR code (line 38 – 39), which retrieves the issuance request object from the request cache when scanned via the contained URL. The issuance request is encoded as a JWT and contains all the information that the app can use to retrieve the actual VC from Azure. The addressed routes for such interactions originating from the Authenticator app towards the reference implementation are implemented similarly to the other solutions in the form of express routes in the `AzureUtilRoutes.ts`.

This subsection covered all the specifics of the Azure solution and its integration. The next section summarizes the results of the implementation in more detail.

4.6 Results

Having covered the implemented solutions Mattr, Trinsic, Veramo, and Azure in the last section, the results are now considered with respect to the coverage of the VC lifecycle and some summary conclusions are drawn. Furthermore, the scope of the implementation with respect to unimplemented parts of the individual solutions will be considered in order to provide an overall picture. A summary of all facets and approaches of the solutions with a final score can be found in table 4.2.

Table 4.2: Implementation results

Step	Feature	Mattr	Trinsic	Veramo	Azure
Issue VC	**Implemented**	✓	✓	✓	✓
	Direct	✓	✗	✓	✗
	QR code	✓	✓	✗[2]	✓
	Comm	✓[1]	✓[1]	✓	✗
Store VC	**Implemented**	✗	✗	✓	✗
	Direct	✗	✗	✓	✗
	Indirect	✓	✓	✓	✓
Transfer VC	**Implemented**	✗	✗	✓[3]	✗
Compose VP	**Implemented**	✓	✗	✓	✗
	Direct	✓	✗	✓	✗
	Indirect	✓	✓	✗[2]	✓
Present VP	**Implemented**	✗	✗	✓	✗
	Direct	✗	✗	✓	✗
	Indirect	✓	✓	✗[2]	✓
Request VP	**Implemented**	✓	✓	✓	✓
	QR	✓	✓	✗[2]	✓
	Comm	✓[1]	✓[1]	✓	✗
Verify VC/ VP	**Implemented**	✓	✗	✓[3]	✗
	Direct	✓	✗	✓[3]	✗
	Indirect	✓	✓	✓[3]	✓
Revoke VC	**Implemented**	✓	✓	✓[3, 4]	✗
Delete VC	**Implemented**	✓	✓	✓	✓
Derive VC	**Implemented**	✗	✗	✗	✗
	Indirect	✓	✓	✗	✗
Score		75%	65%	75%	60%

[1] Not implemented
[2] Implementation with personal effort possible
[3] Implemented with personal effort using API
[4] Restricted to did:ethr

The score is calculated as follows: If a process step could be implemented directly using the available API, one point is awarded. Half a point is awarded if the process step is indirectly contained in another process step and cannot be implemented independently via the API. Half a point is also awarded for process steps that had to be implemented by the author using available API methods in order to be represented. If the process step could not be implemented, 0 points are awarded. At the end, the points received are added up and displayed as a percentage of all possible points. The individual steps were not weighted with regard to different requirements for different use cases.

To show the differences in the implementation in table 4.2 in a more granular way, there are some refinements added to some steps. `Direct` refers to the fact that in this step a JSON LD object, for example in the form of a VC, can be used directly as an input. In contrast, `Indirect` refers to the fact that this process step is part of another process step. For example, in MATTR, the store step could not be implemented directly, but is indirectly part of the issuance step, where the created VC is stored in the cloud agent. This distinction is helpful to show that many unimplemented steps are nevertheless represented by the solutions, but were often abstracted away. Finally, `Comm` and `QR code` are used to document different ways to interact with other agents/wallets. For example, in MATTR, a VC can be transmitted to a wallet both by scanning a QR code but also by sending it directly through a communication module. Considering the scope of the work, not all facets of the individual solutions were necessarily implemented, which is why unimplemented features were marked accordingly. In addition, other annotations concerning (possible) self-implementations and restrictions were added as well.

From these results it is apparent that none of the solutions can directly map the complete VC lifecycle. All of them cover the steps issuance, presentation request, and credential deletion, but none of the solutions offers a direct method to derive VCs for ZKP purposes. The different approaches taken by the solutions are also revealed here. MATTR and Veramo give developers more freedom and abstract fewer complexities than Trinsic and Azure. This is noticeable in the fact that it is possible to work directly with JSON-LD objects from credentials and more functionalities are exposed through their APIs. This allows Mattr and Veramo to achieve a score of 75%. Nevertheless, it must be clearly distinguished that Mattr abstracts and restricts more than Veramo in some areas, which results however in more finished and production-ready APIs and tools. Especially in this area, Veramo is still lagging behind, but its high flexibility and openness makes it possible to retrofit necessary features with existing API methods with manageable effort. For example, Veramo's verify method is incomplete, as described earlier, where missing checks can be implemented by the user. This also allows, for example, that with the help of the communication module in Veramo the transfer step could be implemented as the only solution. This is not possible with Mattr. A potential developer must clearly assess what is important to its use case.

In contrast, Trinsic and Azure abstract away significantly more complexities and functionality, but this can be accompanied by a faster and simpler implementation in some cases. In addition, no interaction with VCs as raw JSON-LD objects is possible here at any point. This ensured that Trinsic and Azure could only achieve 65% and 60%. To further rank the solutions, the following table 4.3 provides a quick breakdown of some of the features and standards present.

Table 4.3: Rough feature comparison

Feature	**MATTR**	**Trinsic**	**Veramo**	**Azure**
Format	JSON-LD	JSON	JSON-LD	JSON-LD
Proof	LD Proofs	JWT	JWT	JWT
ZKP	BBS+	CL Signatures	✗	✗
Revocation	RevocationList 2020	Indy Revocation Registry	EthrStatus Registry2019	Proprietary
Messaging	JWM	DIDComm v1	DIDComm v2	✗
DID Methods	key, web, sov, ion	sov, peer	key, web, ethr, ion	ion

With regard to related features, it is noteworthy that Mattr is the only solution that supports LD proofs, various DID methods, BBS+ and RevocationList2020 as a DID method independent revocation method. In the messaging area, only the precursor to DIDComm (JSON Web Message) is supported. Technologically, none of the other solutions can match Mattr at the current stage. Veramo only supports JWT as a proof format, which is why advanced ZKP technologies such as BBS+ cannot be supported. Furthermore, there is currently only one library that can be used to revoke did:ethr credentials with the EthrStatusRegistry2019 method. Nevertheless, many DID methods are supported and a DIDComm v2 implementation is usable. In contrast, Trinsic currently has no support for JSON-LD, LD proofs and the benefits built on top of them. The Hyperledger link becomes apparent here, since only its technologies such as CL signatures, DID methods, revocation format, and messaging protocol are supported. However, as described in subsection **??**, the current beta of Trinsic includes many of these new technologies, which could affect Trinsic's score in the future. Announcements from Veramo indicate similar plans. Azure seems to be quite behind at this point, with not supporting standards or only using proprietary ones.

Now that a basic examination of the individual solutions and their features has been made, the next chapter will present a new developer-oriented evaluation framework based on these observations, which could be used to evaluate solutions for SSI based on a defined set of criteria. For this purpose, the observation from table 4.3 will also be expanded further.

5 Evaluation Framework

Allgemeiner Einstieg; Warum noch mal eval frame? Ziel.

5.1 Requirements

Grundlegende Anforderungen -> developer focus, was soll abgebildet werden (siehe implementierung, umfrage)

5.2 Criteria & Questions

Prozess: Grundlegende Bereiche/ Kategorien definieren (als Frage), definieren der Kriterien -> Fragen

Darstellung in Tabellenform

Punktevergabesystem

5.3 Results

Einordnen der Lösungen in Tabelle -> Bewertung

Was sind die Ergebnisse? Welche Lösung eignet sich für das? Was machen welche Lösungen sehr gut, sehr schlecht? Überraschungen bezügluch Vorbetrachtung?

6 Conclusion

Bibliography

[Al16] Allen, Christopher: , The Path to Self-Sovereign Identity, April 2016.

[AL20] Allende López, Marcos: Self-Sovereign Identity: The Future of Identity: Self-Sovereignity, Digital Wallets, and Blockchain. Inter-American Development Bank, September 2020.

[BC18] Borgogno, Oscar; Colangelo, Giuseppe: Data Sharing and Interoperability Through APIs: Insights from European Regulatory Strategy. SSRN Electronic Journal, 2018.

[Be19] Bernabe, J. Bernal; Canovas, J. L.; Hernandez-Ramos, J. L.; Moreno, R. Torres; Skarmeta, A.: Privacy-Preserving Solutions for Blockchain: Review and Challenges. IEEE Access, 7:164908–164940, 2019. Conference Name: IEEE Access.

[Bo12] Borchers, Detlef: , Der Diginotar-SSL-Gau und seine Folgen, January 2012.

[Bo19] van Bokkem, Dirk; Hageman, Rico; Koning, Gijs; Nguyen, Luat; Zarin, Naqib: Self-Sovereign Identity Solutions: The Necessity of Blockchain Technology. arXiv:1904.12816 [cs], April 2019. arXiv: 1904.12816.

[Bo20] Bouras, Mohammed Amine; Lu, Qinghua; Zhang, Fan; Wan, Yueliang; Zhang, Tao; Ning, Huansheng: Distributed Ledger Technology for eHealth Identity Privacy: State of The Art and Future Perspective. Sensors, 20(2):483, January 2020.

[Bu20] Bundesdruckerei: , So funktionieren digitale Identitäten, March 2020.

[Ca05] Cameron, Kim: , The Laws of Identity, May 2005.

[CK01] Clauß, Sebastian; Köhntopp, Marit: Identity management and its support of multilateral security. Computer Networks, 37(2):205–219, October 2001.

[Da21] Davie, Matthew; Gisolfi, Dan; Hardman, Daniel; Jordan, John; O'Donnell, Darrell; Reed, Drummond; Deventer, Oskar van: , 0289: The Trust Over IP Stack, May 2021.

[Di21] Digital Bazaar: , vc-revocation-list-2020, May 2021. original-date: 2020-04-21T21:56:17Z.

[DP18] Dunphy, Paul; Petitcolas, Fabien A.P.: A First Look at Identity Management Schemes on the Blockchain. IEEE Security & Privacy, 16(4):20–29, July 2018.

[DT20] Dib, Omar; Toumi, Khalifa: Decentralized Identity Systems: Architecture, Challenges, Solutions and Future Directions. Annals of Emerging Technologies in Computing, 4(5):19–40, December 2020.

[Eh21] Ehrlich, Tobias; Richter, Daniel; Meisel, Michael; Anke, Jürgen: Self-Sovereign Identity als Grundlage für universell einsetzbare digitale Identitäten. HMD Praxis der Wirtschaftsinformatik, February 2021.

[FCA19] Ferdous, Md Sadek; Chowdhury, Farida; Alassafi, Madini O.: In Search of Self-Sovereign Identity Leveraging Blockchain Technology. IEEE Access, 7:103059–103079, 2019.

[Ga95] Gamma, Erich, ed. Design patterns: elements of reusable object-oriented software. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass, 1995.

[GMM18] Grüner, Andreas; Mühle, Alexander; Meinel, Christoph: On the Relevance of Blockchain in Identity Management. arXiv:1807.08136 [cs], July 2018. arXiv: 1807.08136.

[Ha18] Hardman, Daniel: , 0011: Credential Revocation, 2018.

[Ha19] Hardman, Daniel: , Aries RFC 0005: DID Communication, November 2019. original-date: 2019-05-08T16:49:20Z.

[Ha21] Hardman, Daniel: , DIDComm Messaging Specification, 2021.

[He07] Hevner, Alan R: A Three Cycle View of Design Science Research. 19:7, 2007.

[He20a] Heck, Rouven: , SSI Architecture Stack, 2020.

[He20b] Helmy, Nader: , JWT vs Linked Data Proofs: comparing VC assertion formats, August 2020.

[He20c] Helmy, Nader: , A solution for privacy-preserving verifiable credentials, August 2020.

[Ho20] Homeland Security: , Preventing Forgery & Counterfeiting of Certificates and Licenses – Phase 1 Interoperability Plug Fest Test Plan, May 2020.

[Ho21a] Hollington, Jesse: , In a Surprising Twist, Apple Just Launched a Tool to Transfer iCloud Photos to Google Photos (But There's a Catch), March 2021. Section: News.

[Ho21b] Homeland Security: , Interoperability Plugfest #2 – VC/DID Multi-Platform/Multi-Vendor Interoperability Showcase/Demo, March 2021.

[Hu14] Hub Culture: , HubID First to Deploy Windhover Principles and Framework for Digital Identity, Trust and Open Data, October 2014.

[Hy18] Hyperledger: , How Credential Revocation Works — Hyperledger Indy SDK documentation, 2018.

[id14] idcubed.org: , ID3 - idcubed.org - The Windhover Principles for Digital Identity, Trust, and Data, November 2014.

[JHF03] Jordan, Ken; Hauser, Jan; Foster, Steven: The Augmented Social Network: Building identity and trust into the next-generation Internet. First Monday, 8(8), August 2003.

[Jo20] John, Anil: , DHS SVIP Blockchain/DLT/SSI Cohort - Multi-Product Phase 1 Interop Artifacts/ Scaffolding / Information, December 2020.

[Jo21] Johnson, Joseph: , Internet users in the world 2021, July 2021.

[Ka15] Kalodner, Harry; Carlsten, Miles; Ellenbogen, Paul; Bonneau, Joseph; Narayanan, Arvind: , An empirical study of Namecoin and lessons for decentralized namespace design, 2015.

[Ko21] Koppenhöfer, Laura: , Kabinettsbeschluss: Handy-Personalausweis ab September, October 2021.

[Kr19] Krempl, Stefan: , E-Government-Studie: Bundesbürger nutzen Personalausweis mit eID kaum, October 2019.

[Ku20] Kuperberg, Michael: Blockchain-Based Identity Management: A Survey From the Enterprise and Ecosystem Perspective. IEEE Transactions on Engineering Management, 67(4):1008–1027, November 2020.

[Li20] Liu, Yang; He, Debiao; Obaidat, Mohammad S.; Kumar, Neeraj; Khan, Muhammad Khurram; Raymond Choo, Kim-Kwang: Blockchain-based identity management systems: A review. Journal of Network and Computer Applications, 166, September 2020.

[Lo20] Lomas, Natasha: , Facebook's photo porting tool adds support for Dropbox and Koofr, March 2020.

[LS21a] Longley, Dave; Sporny, Manu: , Revocation List 2020, April 2021.

[LS21b] Looker, Tobias; Steele, Orie: , BBS+ Signatures 2020, June 2021.

[Ma12] Marlinspike, Moxie: , What is "Sovereign Source Authority"?, February 2012.

[MA20a] MATTR: , Adding support for revocation of Verifiable Credentials, October 2020.

[MA20b] MATTR: , Intro to ZKPs using BBS+ signatures, July 2020.

[MA21a] MATTR: , Approach - Open Source & Interoperable Digital Trust | MATTR, 2021.

[MA21b] MATTR: , MATTR, 2021.

[MA21c] MATTR: , MATTR - YouTube, 2021.

[MA21d] MATTR: , MATTR | Restoring Trust in Digital Interactions with Decentralized Identity, 2021.

[MA21e] MATTR: , MATTR Learn | Docs & API References – Decentralized Identity, 2021.

[MA21f] MATTR: , MATTR Uses A Pay-As-You-Go Pricing Model | MATTR Learn, 2021.

[MA21g] MATTR: , MATTR VII Platform Overview | MATTR Learn, 2021.

[MA21h] MATTR: , Privacy Policy | MATTR Learn, July 2021.

[MA21i] MATTR: , Products - Decentralized Identity Solution | MATTR, 2021.

[MA21j] MATTR: , Resources - Articles and Core Concepts about Decentralized Identity | MATTR, 2021.

[MA21k] MATTR: , VII Core - Component Overview | MATTR Learn, 2021.

[MA21l] MATTR: , VII Core - DIDs (Decentralized Identifiers) | MATTR Learn, 2021.

[MA21m] MATTR: , VII Drivers - Pluggable & Future-proof Integrations | MATTR Learn, 2021.

[MA21n] MATTR: , VII Extensions - Pre-Built Extensions | MATTR Learn, 2021.

[Ma21o] Mayer, Peter; Zou, Yixin; Schaub, Florian; Aviv, Adam J: “Now I’m a bit angry:” Individuals’ Awareness, Perception, and Responses to Data Breaches that Affected Them. USENIX Security Symposium, 30:18, 2021.

[MG20] Meinel, Christoph; Gayvoronskaya, Tatiana: Blockchain: Hype oder Innovation. Springer Berlin Heidelberg, Berlin, Heidelberg, 2020.

[Mi20] Minor, Jens: , Google Fotos: Praktisches Export-Werkzeug - so lassen sich alle Facebook-Fotos & Videos zu Google übertragen - GWB, December 2020.

[Mi21] Microsoft: , Identitätsnachweis-Lösungen – Microsoft Security, 2021.

[NJ20] Naik, Nitin; Jenkins, Paul: uPort Open-Source Identity Management System: An Assessment of Self-Sovereign Identity and User-Centric Data Platform Built on Blockchain. In: 2020 IEEE International Symposium on Systems Engineering (ISSE). IEEE, Vienna, Austria, pp. 1–7, October 2020.

[No21] Noor, Poppy: , Should we celebrate Trump's Twitter ban? Five free speech experts weigh in, January 2021. Section: US news.

[NQ21] Neira, Barclay; Queern, Caleb: , Introduction to Azure Active Directory Verifiable Credentials (preview), January 2021.

[Op21a] OpenJS Foundation: , About Node.js, 2021.

[Op21b] OpenJS Foundation: , Express - Node.js web application framework, 2021.

[OR20] Oesch, Sean; Ruoti, Scott: That Was Then, This Is Now: A Security Evaluation of Password Generation, Storage, and Autofill in Browser-Based Password Managers. In: USENIX Security Symposium. pp. 2165–2182, 2020.

[Or21] Ormandy, Tavis: , Password Managers., June 2021.

[PR21] Preukschat, Alex; Reed, Drummond: Self-Sovereign Identity: Decentralized digital identity and verifiable credentials. Manning, 1 edition, May 2021.

[Ri21a] Riedel, Martin: , Generalized Verification API in Veramo Core #375, August 2021.

[Ri21b] Riley Hughes: , Announcing Trinsic's Largest Platform Update Ever, July 2021.

[Ri21c] Risk Based Security: , Data Breach QuickView - June 2021, July 2021. Section: Featured.

[Si21] Simons, Alex: , Announcing Azure AD Verifiable Credentials, April 2021. Section: Azure Active Directory Identity Blog.

[SLC19] Sporny, Manu; Longley, Dave; Chadwick, David: , Verifiable Credentials Data Model 1.0, November 2019.

[SNA21] Soltani, Reza; Nguyen, Uyen Trang; An, Aijun: A Survey of Self-Sovereign Identity Ecosystem. Security and Communication Networks, 2021:1–26, July 2021.

[SNE20] Satybaldy, Abylay; Nowostawski, Mariusz; Ellingsen, Jørgen: Self-Sovereign Identity Systems: Evaluation Framework. In (Friedewald, Michael; Önen, Melek; Lievens, Eva; Krenn, Stephan; Fricker, Samuel, eds): Privacy and Identity Management. Data for Better Living: AI and Privacy, volume 576, pp. 447–461. Springer International Publishing, Cham, 2020. Series Title: IFIP Advances in Information and Communication Technology.

[Sp21] Sporny, Manu; Longley, Dave; Sabadello, Markus; Reed, Drummond; Steele, Orie; Allen, Christopher: , Decentralized Identifiers (DIDs) v1.0, March 2021.

[St17] Steel, Amber: , LastPass Reveals 8 Truths about Passwords in the New Password Exposé, November 2017.

[St21] Strüker, Dr Jens; Urbach, Dr Nils; Guggenberger, Tobias; Lautenschlager, Jonathan; Ruhland, Nicolas; Sedlmeir, Johannes; Stoetzer, Jens-Christian; Völter, Fabiane: Grundlagen, Anwendungen und Potenziale portabler digitaler Identitäten. p. 52, June 2021.

[Sw21] Swinhoe, Dan: , The 15 biggest data breaches of the 21st century, January 2021.

[TCSL21] Terbu, Oliver; Curran, Stephen; Snorre Lothar, von Gohren Edwin: , Tracking library polooza, September 2021.

[To17] Tobin, Andrew; Reed, Drummond; Windley, Foreword Phillip J; Foundation, Sovrin: The Inevitable Rise of Self-Sovereign Identity. p. 24, 2017.

[To21] Toth, Marek: , You should turn off autofill in your password manager | Marek Tóth, July 2021.

[Tr21a] Trinsic: , Identity Wallets, 2021.

[Tr21b] Trinsic: , Introduction, 2021.

[Tr21c] Trinsic: , Open Source, 2021.

[Tr21d] Trinsic: , Pricing, 2021.

[Tr21e] Trinsic: , Service Clients (SDKs), 2021.

[Tr21f] Trinsic: , Trinsic - A full-stack self-sovereign identity (SSI) platform, 2021.

[Tr21g] Trinsic: , Trinsic Core - Trinsic, 2021.

[Tr21h] Trinsic: , Trinsic Ecosystems, 2021.

[Tr21i] Trinsic: , Trinsic Studio, 2021.

[uP21a] uPort: , uPort, 2021.

[uP21b] uPort: , Veramo: uPort's Open Source Evolution, May 2021.

[Ve21a] Veramo: , Blog | Performant and modular APIs for Verifiable Data and SSI, 2021.

[Ve21b] Veramo: , uport-project/veramo-plugin, September 2021. original-date: 2020-11-19T16:13:43Z.

[Ve21c] Veramo: , Veramo - A JavaScript Framework for Verifiable Data | Performant and modular APIs for Verifiable Data and SSI, 2021.

[Ve21d] Veramo: , Veramo Agent | Performant and modular APIs for Verifiable Data and SSI, 2021.

[WH07] Wilde, Thomas; Hess, Thomas: Forschungsmethoden der Wirtschaftsinformatik. p. 8, 2007.

[WO01] Wilcox-O'Hearn, Zooko: , Names: Distributed, Secure, Human-Readable: Choose Two, October 2001.

[Wo17] World Bank: , 1.1 Billion 'Invisible' People without ID are Priority for new High Level Advisory Council on Identification for Development, 2017.

[Wo21a] World Wide Web Consortium Credentials Community Group: , VC HTTP API, August 2021. original-date: 2020-08-14T00:09:39Z.

[Wo21b] World Wide Web Consortium Credentials Community Group: , Verifiable Credentials HTTP API v0.3, July 2021.

[Ye21] Yegupov, Konstantin: , The demo app's SDK usage does not match the SDK types #38, June 2021.

[Yi21] Yildiz, Hakan: , Layers of SSI Interoperability, January 2021.

[Yo21] Young, Kaliya: Verifiable Credentials Flavors Explained. p. 21, 2021.

[Za20] Zammetti, Frank: Modern Full-Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker. Apress, Berkeley, CA, 2020.

[Zu21] Zundel, Brent: , Why the Verifiable Credentials Community Should Converge on BBS+, March 2021. Section: Thought Leadership.

A Appendix