UNIVERSITÀ DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

FINAL DISSERTATION

Design and implementation of a verifiable credentials service for a data marketplace

Supervisor

Prof. Silvio Ranise

Student

Rupert Gobber

Co-supervisors

Dr. Giada Sciarretta

Dr. Alessandro Tomasi

Academic year 2020/2021

Acknowledgments

This thesis would not have been feasible without the invaluable help of some people whom I want to thank. First of all, I thank Prof. Silvio Ranise for giving me the opportunity to carry out this thesis in collaboration with the Security and Trust unit of the Fondazione Bruno Kessler research center. I especially want to thank Dr. Giada Sciarretta and Dr. Alessandro Tomasi for their support in writing the thesis with valuable comments that have always helped me.

I also want to thank my colleagues at GFT, in particular Luca Marangoni for the technical support in the implementation and Alessandro Amicone for the great project management within i3Market and for allowing me to work in this interesting and innovative project.

Last but not least, I thank my parents Luigina and Ivo and my brother Alois who have been with me since the beginning of my journey and all my friends for their constant support.

Contents

Abstract

In recent years, the Internet has been a revolution that has enabled the rise of many online services around the world. It was a disruptive technology that helped shape modern society as we know it today, allowing many businesses to reach an unknown large audience. However, trusting a person or company with whom we interact but we are unable to physically see is a major problem. To ensure trust in such huge network, several digital identity models have been experimented over the years in an attempt to securely authenticate people. Current centralized and federated models have not been designed for a user-centered approach, and as the importance of digital identity increases, they have begun to present two major problems. First, if each organization uses its own identity system to authenticate its users, users' identities are fragmented across many different systems, forcing the user to create identities for all the organizations they are interacting with. The second problem is the aggregation of personal information in the databases of large IT companies, forcing users to place an enormous amount of trust in someone else to keep our information private and secure.

Self-sovereign identity is an emerging identity model that uses cryptography and distributed ledger technologies to solve these problems. The goal of this digital identity model is to return digital identities in the hands of users by placing them at the center of the model. Grouping all their personal information under a single identity would improve the user experience and allow for better control over how personal data is used.

This thesis derives from the experience gained in the H2020 i3-Market project, which aims to create an open source data marketplace for the sale and purchase of datasets. In this marketplace, users can purchase assets and services by revealing verifiable credentials, that contains limited attributes of their identity, taking advantage of the benefits of self-sovereign identity, avoiding the centralized grouping of user information. This thesis describes the design and implementation of the service in i3-Market to manage the verifiable credentials of the users using the uPort framework and the integration with the other actors foreseen in a self-sovereign identity ecosystem.

1 Introduction

Today more than ever online services are used and will continue to be so in the future. The number of actors we are interacting with has also significantly increased. Every day we interact with friends, colleagues, strangers, private companies that provide services or products and also with public services. Currently, the identity models we are using are mostly centralized or federated. These models do not allow for the optimal management of digital identity. In fact, users have to rely on third-party systems and have no insight into how their data is used and whether it is safe. Very often, without realizing it, we grant permissions for the processing of this personal data which are then sold to companies that use them in targeted marketing campaigns. Furthermore, very often we are forced to have to update multiple accounts with outdated information, making us waste time and above all control over our data. For example, if you move home it is necessary to update the Amazon shipping address, or if you receive magazines at home it becomes necessary to update the delivery address and so on for many other types of services.

Nowadays there are some technologies like OpenID, Google, Facebook Connect, Twitter Connect that allow users to access websites that have integrated their solutions using a single account, however, users still have to create their identities through organizations. Relying on central authorities without data protection is risky as these organizations could misuse your identity without you noticing or be hacked and end up in the hands of potentially dangerous actors. Already in the past, big tech companies have often undergone important data breaches that have compromised the confidentiality of the personal data of millions of people [26] [22].

Trusting people and having a safe interaction in a digital ecosystem is a problem that we have been trying to solve for many years. Trust is often related to issues such as security, respect for user privacy, technical reliability or the authenticity of information obtained on the Internet.

To make the identity of users decentralized it was necessary to think of a new model for its management. In this new model, the user must be at the center of the system and be the one and only owner of his identity. To explain the identity of self-sovereignty we will first examine its principles. In one of his essays, Christopher Allen introduces a list of ten principles to help understand what self-sovereign identity is [1].

- Existence: Users must have an independent existence.
- Control: Users must control their identities.
- Access: Users must have access to their own data.
- Transparency: Systems and algorithms must be transparent.
- Persistence: Identities must be long-lived.
- Portability: Information and services about identity must be transportable.
- Interoperability: Identities should be as widely usable as possible.
- Consent: People must freely agree to how their identity information will be used.
- Minimalization: Disclosure of claims about an identity must be as few as possible.
- Protection: The rights of individual people must be protected against the powerful.

This list of principles is also a guide for organizations and individuals as they attempt to return control over digital identity and personal data to their owner. For many years proving our identity online as we do it offline has been a problem. This is because Internet is an environment of distrust and the only available solution to add some trust has been to rely on mechanisms such as public key infrastructure (PKI). Few years ago, we had no ways to create an open PKI and we had to rely on centralized authorities to keep them secure. However, these centralized authorities represent central points of failure. Anyone able to access them could compromise the integrity of the entire internet. The blockchain technology used in self-sovereign identity's architecture enables us to create the secure and open PKI that we need to redesign the entire concept of digital identity. The applications it can provide are multiple. Using this technology there is the possibility to build the identity management layer which is missing to the Internet and provide an universal identity for its users. This identity is owned by the entity it represents and for this reason it respect its privacy. This identity model allows the collection and exchange of verifiable credentials with the possibility to verify digital signatures of their issuers.

1.1 Thesis outline and contributions

This thesis first describes the state of the art of digital identity models and how centralized and federated models have evolved, in order to understand what are the problems that have arisen and how the self-sovereign identity model aims to solve them. Chapter 2 presents the background on recent standards and protocols that allow authentication in federated models, then it presents an overview of the technologies used to develop the solution described Chapter 4 and 5. Chapter 3 is aimed at giving a context to the project on which this thesis contributes. In particular, it describes at a high level how the i3-Market framework works and how the solution described in this thesis is used in relation to the other components and actors of the system. Chapter 4 is dedicated to the description of the component developed to manage the verifiable credentials of users in a self-sovereign identity context. In particular, it describes how and when verifiable credentials are issued, verified and revoked through i3-Market use cases. Chapter 5 describes how the other actors of the self-sovereign identity ecosystem integrate with the component developed to manage verified credentials. Finally, Chapter 6 presents the conclusions and knowledge gained by experimenting with this model for digital identity and these technologies, opportunities for future work and the future challenges that this model will face.

By designing and implementing a verifiable credential management service and its integration with the other actors of the self-sovereign identity ecosystem, this thesis aims to contribute to the expansion and development of further solutions, demonstrating that this identity model is effective in the i3-Market project but could also be effective in many other scenarios yet to be explored.

2 Background

This chapter presents state-of-the-art technologies and the most widely used standards for identity solutions. Section 2.1 introduces the main concepts about OAuth. Section 2.2 presents JSON Web Token and how these standardized access tokens ensure the secure exchange of data between two parties. Section 2.3 describes the state-of-the-art technologies that implement distributed identity. Section 2.4 presents an overview of verifiable credentials and their ecosystem. Section 2.5 introduces the uPort technology, the main concepts and the libraries used to manage verifiable credentials. Section 2.6 presents the Veramo framework, another framework used to manage verifiable credentials.

2.1 OAuth

The OAuth 2.0 authorization framework is a protocol that allows a user to grant a third-party web site or application access to the user's protected resources [4] [12]. In this framework, a client requests access to resources controlled by the resource owner and hosted by the resource server. Instead of using the resource owner's credentials to access protected resources, the client obtains an access token, and optionally, a refresh token. An access token is a string denoting a specific scope, lifetime, and other attributes. Since access tokens have a lifetime, they can expire. The refresh tokens contain the information needed to obtain a new access token. Therefore, whenever an access token is required to access a specific resource and the access token has expired, the client can use a refresh token to obtain a new access token issued by the authorization server. The access token is used by the client application to access the protected resources hosted by the resource server. When an application authenticates to the third-party web site or application, it specifies the scopes that are needed for that application. If the user authorizes those scopes, then the access token will represent these authorized scopes and it can be used to access the protected resources.

OAuth Entities
In the OAuth 2.0 protocol there are five entities involved:

- Resource Owner: it is the entity that can grant access to a protected resource. Typically, this is the end-user.
- Client: it is the application requesting access to a protected resource on behalf of the Resource Owner. Typically, this is a single page application, a native or a web application.
- Resource Server: it is the server hosting the protected resources and exposes them via APIs.
- Authorization Server: it is the server that authenticates the Resource Owner and issues access tokens.
- User Agent: agent used by the Resource Owner to interact with the Client. Typically a browser or a native application.

In general, an authorization process can be described as follows:

- The Client application sends an authorization request to the Authorization Server to access a protected resource
- The Resource Owner, i.e. the user, grants the access to the resource
- The Authorization Server returns an access token to be used in all subsequent requests.

Going into detail, this process differs according to the type of grant used. A grant type is a flow to get an access token. There are different types of grants, suitable for different purposes. Deciding which is the most suitable depends mainly on the type of application, but other parameters also matter, such as the level of trust for the customer or the user experience of the application. There are four grant types in OAuth 2.0. They are as follows:

- Authorization Code Flow: it is the grant most widely used by web applications executing on a server. It is also used by mobile applications, using the Proof Key for Code Exchange (PKCE) technique (described below).
- Implicit Flow with Form Post: simplified version of the authorization code flow where instead of issuing an authorization code, the client is issued an access token directly.
- Resource Owner Password Flow: grant that can be used in highly-trusted applications, because it requires sharing user credentials with the app.
- Client Credentials Flow: grant used for machine-to-machine communication, where the client is also the resource owner.

Authorization Code Flow with Proof Key for Code Exchange (PKCE)
The Authorization Code grant is the most used and recommended flow by OAuth 2.0 security best practices [18]. To use this flow, the application must have been previously registered with the Authorization Server. To register a client application on the authorization server, usually only a few parameters need to be provided. In particular the name of the client application, a redirect URI that will be used to inform the application that the user has authorized the request and another redirect URI for the logout operation, which usually redirects to the page of login. As with all OAuth 2.0 flows, during registration the authorization server assigns to the application a client ID and a client secret, which will be used during the authorization process. To access a specific resource during the authentication flow, the application must indicate the list of required "authorizations", called scopes. Scopes are normally defined by the Resource Server. Based on the scope communicated, the access token will have different permissions to retrieve various resources from the Resource Server. The required steps to perform the authentication process are shown in Figure 2.1.

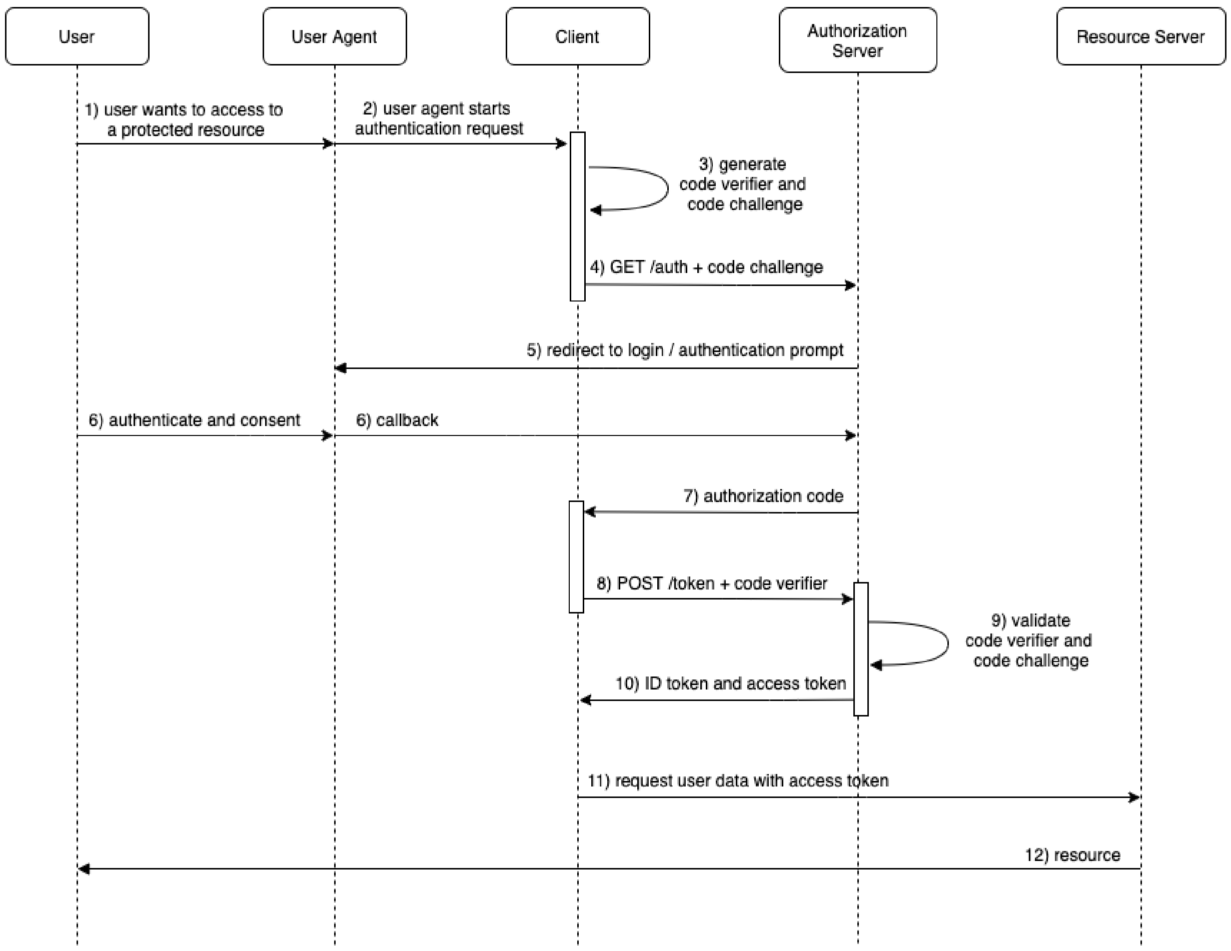

User
User Agent
Client
Authorization Server
Resource Server
1) user wants to access to a protected resource
2) user agent starts authentication request
3) generate code verifier and code challenge
4) GET /auth + code challenge
5) redirect to login / authentication prompt
6) authenticate and consent
6) callback
7) authorization code
8) POST /token + code verifier
9) validate code verifier and code challenge
10) ID token and access token
11) request user data with access token
12) resource

Figure 2.1: Authorization Code Flow with Proof Key for Code Exchange (PKCE)

The flow begins with the user making a login request on the client application (1). The client application creates a cryptographically random *code_verifier* and from this then generates a *code_challenge* (2). The code verifier is a cryptographic random string with a minimum of 43 characters and a maximum of 128 characters. The code challenge is created by SHA256 hashing the *code_verifier*. The resulting hash is then encoded in Base64URL. Usually these parameters are saved in the browser's *localStorage*, which is a key/value data store available in each user's browser. As with cookies, the *localStorage* can only store string data for keys and values. Next, the user is redirected to the authorization server at the endpoint */authorize* (3). In this first call the *code_challenge* must be attached. When the call is received, the authorization server redirects the user to the login and authorization prompt (4). There are many ways to implement the login and authorization prompt. The simplest way is to use username/password but alternatively it is possible to connect external Single Sign On services. The user authenticates using one of the configured access options and can see a consent page that lists the permissions to be granted to the application (5). Once the data for authentication has been entered by the user and the response has been sent back, the authorization server stores the *code_challenge*. Then it redirects the user to the application with an *authorization code* (6), which is valid only for one use and is used to be exchanged with tokens. After the client then receives the *authorization code*, it sends it with the *code_verifier* (previously created in Step 2) to the authorization server, at the endpoint */token* (7). In step (8) the authorization server checks the *code_challenge* and *code_verifier*, thus calculating the hash and verifying the correspondence between them. If nothing went wrong, the authorization server responds with an ID token and an access token (9). The ID token is a security token that contains information about the end user. This information tells to the client application that the user is authenticated, and can also give some information like their username, firstname, lastname, email, and so on. It is a good practice to configure a refresh token as well, for the reasons detailed at the beginning of this section. The client application can use the access token to call the protected APIs (10). The token is usually embedded in the HTTP request header as a bearer token. In Step (11) the API will respond with the requested resource.

PKCE extension

The OAuth2 framework was designed from the ground up to be extensible. In fact, since its beginning, various evolutions have been introduced to deal with potential safety problems or increase its functionality. One of these is PKCE [24], an extension designed to make the authorization code grant more secure and limit the likelihood of the authorization code being intercepted. PKCE does not introduce any new requests, but only provides for the use of additional parameters in standard requests. In this way, clients can still use the PKCE extension even if the server does not support it. The PKCE extension integrates with the standard authorization code flow in the following steps:

- In step (2) of Figure 2.1, the client application generates the code verifier and the code challenge as described before
- In step (3), the code challenge is included in the authorization token generation request
- In step (7), the client application sends the request to generate the access token, including the authorization token and the *code_verifier*
- In step (8), the authorization server, before generating the tokens, verifies that the *code_verifier* corresponds to the *code_challenge* sent previously, applying the hash function

The advantage of this extension is that if the authorization code is intercepted, it cannot be used anyway, because the request for the access token depends on the *code_verifier* generated initially.

2.2 JSON Web Token (JWT)

The JSON Web Token (JWT) is an open standard (RFC 7519) that defines a schema in JSON format for exchanging information between various services [16]. This information can be verified and trusted as it is digitally signed. The token can be signed with a secret key that only those who generate the token know using the HMAC algorithm, or using a pair of public and private keys using the RSA or ECDSA standards. JWTs are widely used to authenticate requests in Web Services and OAuth 2.0. In an authorization scenario, once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Since JWTs can be signed, they are a good way of securely transmitting information between parties, because they guarantee the integrity of the claims, and therefore they can also be used in many other information exchange scenarios. When JWTs are signed using public and private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

The structure of a JWT consists of 3 fundamental parts:

- Header
- Payload
- Signature

The header contains two main information: the type of token (in this case valued to JWT because it is a JSON Web Token) and the type of encryption algorithm used. For example, the following JSON Header declares that the encoded object is a JWT, and the JWT is a JSON Web Signature that is MACed using the HMAC SHA-256 algorithm:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

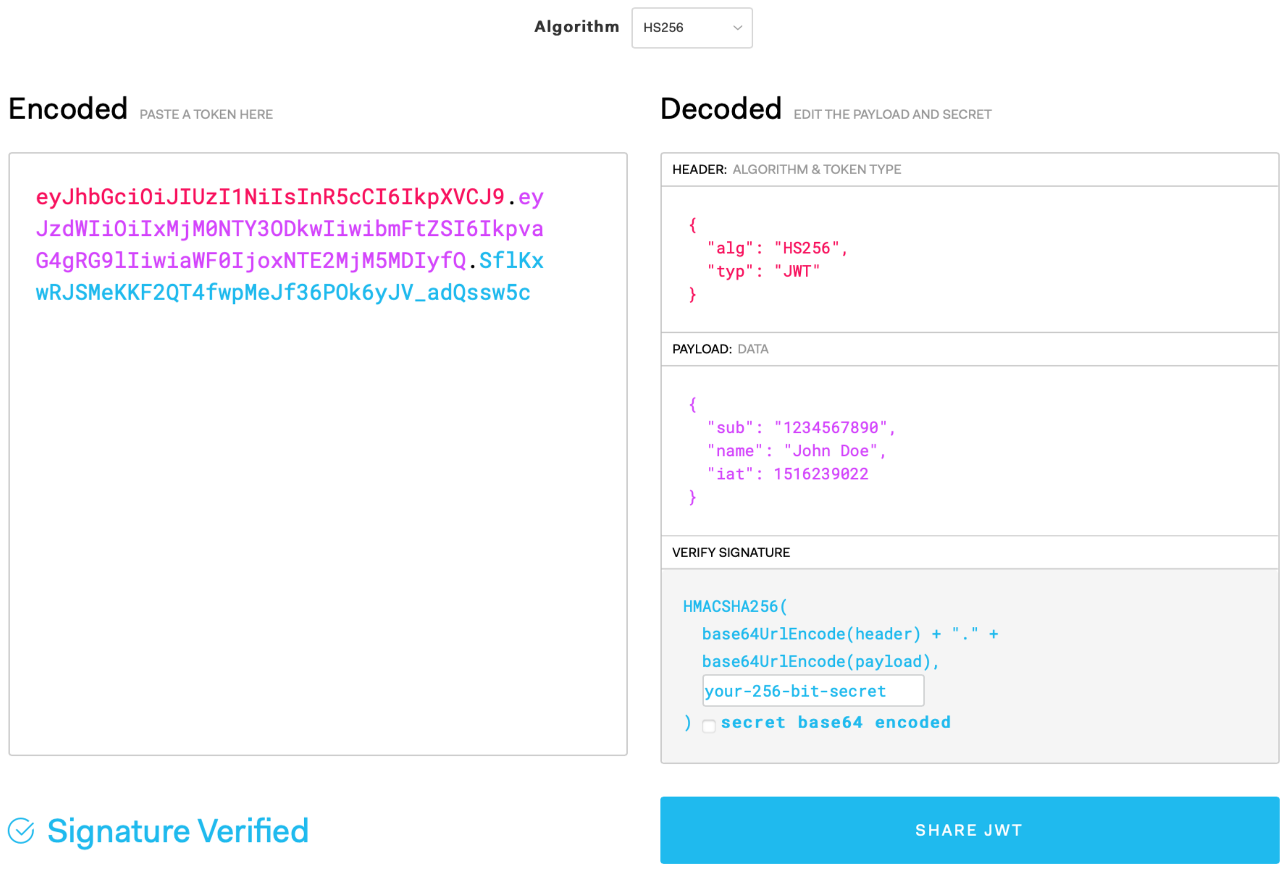
The payload contains the interchange information, i.e., the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

- registered claims: they are predefined properties that indicate information about the token (issuer, audience, expiration, issued at, subject),
- public claims: they refer to claims defined in the IANA JSON Web Token Registry, they can be compiled at will,
- private claims: here it is possible to enter new fields, such as verifiable claims, having full flexibility thanks to the JSON structure.

The JWT Claims Set represents a JSON object whose members are the claims conveyed by the JWT. The set of claims that a JWT must contain to be considered valid is context dependent and is outside the scope of the JWT specification. The next example shows what a JWT payload looks like:

```
{
  "iss": "https://www.jwt-issuer.com",
  "name": "Mario Rossi",
  "iat": 1540890704,
  "exp": 1540918800,
  "user": {
    "profile": "editor"
  }
}
```

The last part of a JWT token is the signature, and it is used to verify the message was not modified. In the case of JWTs signed with a private key, it can also verify that the sender of the JWT is who it says it is. The token is generated by encoding the header and payload in base 64 and joining the two results by separating them by a "." then the algorithm indicated in the header is applied to the string obtained using a secret key. It is possible to verify and unpack a JWT online using the official website [3], that provides the convenient interface shown in Figure 2.2.


Algorithm
HS256
Encoded PASTE A TOKEN HERE
Decoded EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE
PAYLOAD: DATA
VERIFY SIGNATURE
Signature Verified
SHARE JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Figure 2.2: Example of an encoded JWT (left) and its decoding (right)

Fortunately it is not necessary to re-implement the encryption logic because there are many libraries that allow to generate JWT depending on the programming language. In this thesis, the JWT library used to create verifiable credentials and verifiable presentations is the *did-jwt-vc* library [14]. Security is guaranteed by the fact that the JWT is signed with a server-side secret key, so if a JWT is corrupted or modified by an external agent, it will not pass the validation.

2.3 Decentralized identifiers (DIDs)

Decentralized identifiers (DIDs) [30] are a type of identifier that enables verifiable and decentralized digital identity. A DID identifies any subject (e.g., a person, organization, thing, data model, abstract entity, etc.) that the controller of the DID decides that it identifies. In contrast to federated identifiers, DIDs have been designed so that they may be decoupled from centralized registries, identity providers, and certificate authorities. Specifically, while other parties might be used to help enable the discovery of information related to a DID, the design enables the controller of a DID to prove control over it without requiring permission from any other party. DIDs are URIs that associate a DID subject with a DID document allowing trustable interactions associated with that subject. Each DID document can express cryptographic material, verification methods, or services, which provide a set of mechanisms enabling a DID controller to prove control of the DID. Services enable trusted interactions associated with the DID subject. A DID might provide the means to return the DID subject itself, if the DID subject is an information resource such as a data model. Two examples of DID documents are documented in Appendix A.

A DID, as shown in Figure 2.3, is a text string consisting of three parts:

- the DID URI scheme identifier: the DID scheme is the formal syntax of a decentralized identifier. The generic DID scheme begins with the prefix *did:*;
- the identifier for the DID method: it is the definition of how a specific DID method scheme is implemented. In a specific DID method scheme, the DID method name follows the first colon and terminates with the second colon, e.g., *did:example:*;
- the DID method-specific identifier: a DID method is defined by a DID method specification, which specifies the precise operations by which DIDs and DID documents are created, resolved, updated, and deactivated.

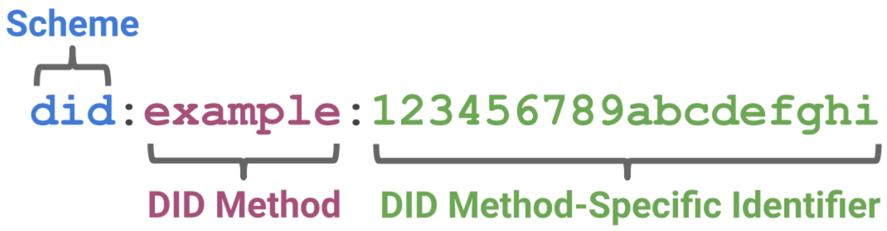

Scheme
did:example:123456789abcdefghi
DID Method
DID Method-Specific Identifier

Figure 2.3: Example of a Decentralized Identifier (DID), from the W3C DID recommendation [30]

As shown in Figure 2.4:

- A Decentralized Identifier (DID) is a URI that refers to a subject.
- DIDs are resolvable to DID documents.
- A DID URL extends the syntax of a basic DID to incorporate other standard URI components such as path, query, and fragment in order to locate a particular resource, for example, a cryptographic public key inside a DID document, or a resource external to the DID document.
- The DID document is controlled by a DID controller, that is the entity (person, organization, or autonomous software) that has the capability to make changes to a DID document. This capability is typically asserted by the control of a set of cryptographic keys used by software acting on behalf of the controller.
- In order to be resolvable to DID documents, DIDs are recorded on an underlying system or network of some kind, called a verifiable data registry. Examples of data registries include distributed ledgers, decentralized file systems, databases of any kind, peer-to-peer networks, and other forms of trusted data storage.

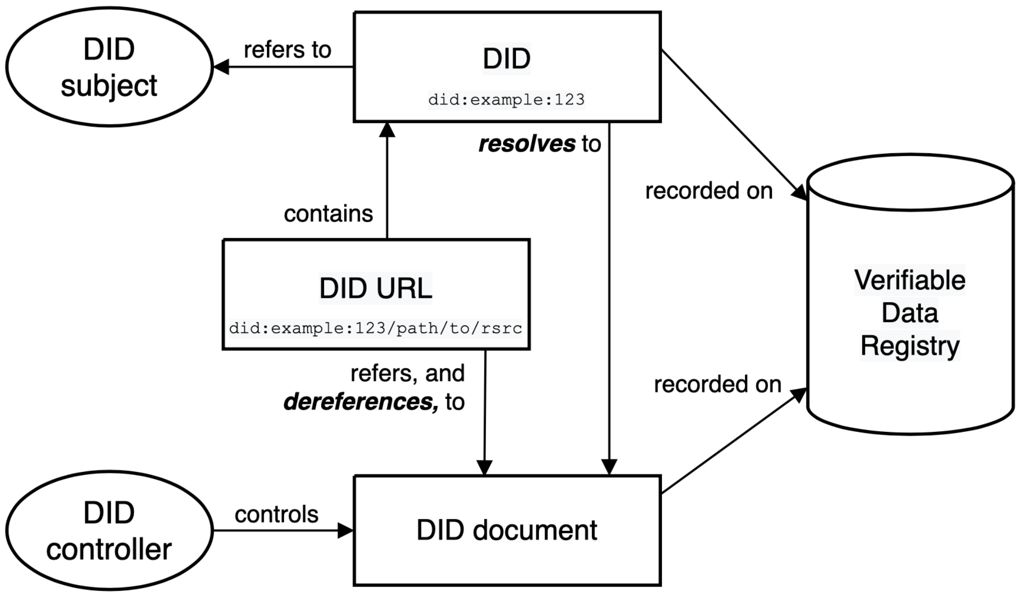


Figure 2.4: DID architecture and the relationship of the basic components, from the W3C DID recommendation [30]

2.4 Verifiable credentials

A verifiable credential is a tamper-evident credential that has authorship that can be cryptographically verified [29]. As in the physical world, a verifiable credential represents a set of information that identifies an entity. In particular, the information are related to:

- the subject of the credential (for example, a photo, name, or identification number)
- the issuing authority (for example, a city government, national agency, or certification body)
- the type of credential (for example, a Dutch passport, an American driving license, or a health insurance card)
- specific attributes or properties being asserted by the issuing authority about the subject (for example, nationality, the classes of vehicle entitled to drive, or date of birth)
- how the credential was derived
- the constraints on the credential (for example, expiration date, or terms of use).

Verifiable Credentials ecosystem
In an ecosystem based on verifiable credentials, several entities can be differentiated playing distinct roles.

- Holder: an entity that possesses one or more verifiable credentials. Examples of holders include students, employees, and customers.
- Subject: an entity about which claims are made. Examples of subjects include human beings, animals, and things.
- Issuer: an entity that asserts claims about one or more subjects, creating a verifiable credential from these claims, and transmitting the verifiable credential to a holder. Examples of issuers include corporations, non-profit organizations, trade associations, governments, and individuals.
- Verifier: an entity that receives one or more verifiable credentials, optionally inside a verifiable presentation, for processing. This entity verifies certain claims and eventually makes decisions from the disclosed information. Examples of verifiers include employers, security personnel, and websites.
- Verifiable data registry: a role a system might perform by mediating the creation and verification of identifiers, keys, and other relevant data, such as verifiable credential schemas, revocation registries, issuer public keys, and so on, which might be required to use verifiable credentials. Examples of verifiable data registries include trusted databases, decentralized databases, government ID databases, and distributed ledgers.

Usually the holder of a verifiable credential is the subject, but in certain cases it is not. For example, a parent (the holder) might hold the verifiable credentials of a child (the subject), or a pet owner (the holder) might hold the verifiable credentials of their pet (the subject). The relationships between roles in an ecosystem based on verifiable credentials is shown in Figure 2.5.

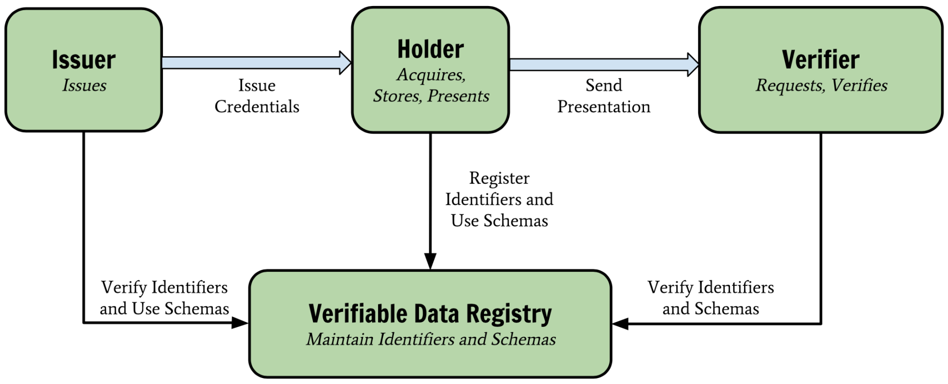

Issuer
Issues
Issue Credentials
Holder
Acquires, Stores, Presents
Send Presentation
Verifier
Requests, Verifies
Register Identifiers and Use Schemas
Verify Identifiers and Use Schemas
Verifiable Data Registry
Maintain Identifiers and Schemas
Verify Identifiers and Schemas

Figure 2.5: Verifiable credentials ecosystem, from the W3C Verifiable Credential data model specification [29]

The main tasks performed by the roles represented in Figure 2.5 are as follows:

- Issuers can issue verifiable credentials about any subject.
- Issuers can revoke verifiable credentials. Revocation by the issuer should not reveal any identifying information about the subject, the holder, the specific verifiable credential, or the verifier.
- Holders can receive verifiable credentials from anyone.
- Holders can store verifiable credentials in any location, without affecting their verifiability and without the issuer knowing anything about where they are stored or when they are accessed.
- Holders assemble collections of verifiable credentials from different issuers into a single artifact, a verifiable presentation. Verifiable credentials represent statements made by an issuer in a tamper-evident and privacy-respecting manner.
- Verifiable presentations are set of verifiable credentials and they allow any verifier to verify the authenticity of verifiable credentials from any issuer.
- Verifiable presentations can either disclose the attributes of a verifiable credential, or satisfy derived predicates requested by the verifier. Derived predicates are Boolean conditions, such as greater than, less than, equal to, is in set, and so on.
- Verifiers can request and verify verifiable credentials from any holder, containing proofs of claims from any issuer.

Verifiable Credentials structure
A credential is a set of one or more claims made by the same entity. A verifiable credential is composed of three basic components, as shown in Figure 2.6:

- Credential Metadata: they include an identifier and metadata to describe properties of the credential, such as the issuer, the expiry date and time, a representative image, a public key to use for verification purposes, the revocation mechanism, and so on. The metadata might be signed by the issuer.
- Claim(s): they are statements about a subject. A subject is an entity about which claims can be made. Claims are expressed using subject-property-value relationships.
- Proof(s): cryptographically proofs that can be used to detect tampering and verify who issued it.

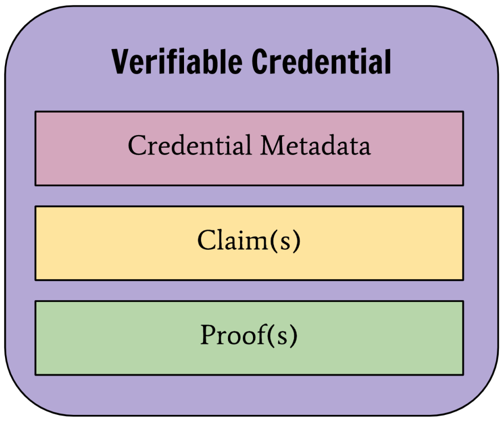


Figure 2.6: Verifiable credentials structure, from the W3C Verifiable Credential data model specification [29]

Figure 2.6 shows the basic components of a verifiable credential, but abstracts the details about how claims are organized into information graphs, which are then organized into verifiable credentials. Figure 2.7 shows a more complete depiction of a verifiable credential, which is normally composed of at least two information graphs. The first graph expresses the verifiable credential itself, which contains credential metadata and claims. The second graph expresses the digital proof, which is usually a digital signature.

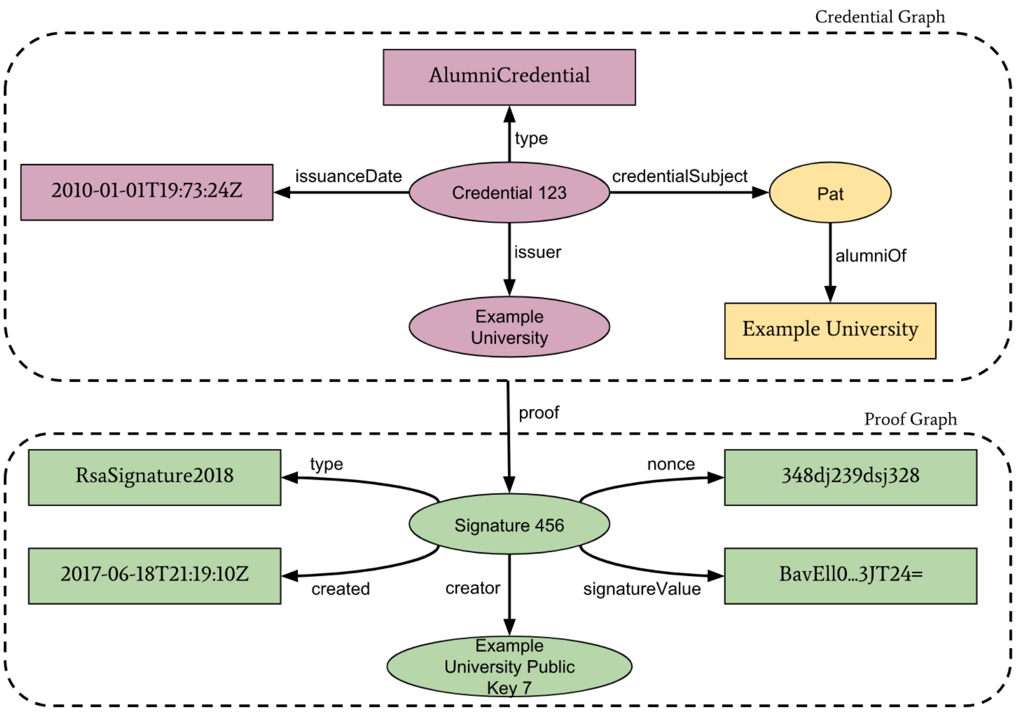


Figure 2.7: Information graphs associated with a basic verifiable credential, from the W3C Verifiable Credential data model specification [29]

As can be seen from the image above, the certificate attribute representing the credential, i.e. the claim, is reported in the *credentialSubject* section. In this case the claim is *"Pat is alumni of ExampleUniversity"*. An example of a verifiable credential is documented in Appendix B. Verifiable credentials can be grouped together, forming a verifiable presentation.

2.5 uPort

uPort is an open source framework based on the Ethereum blockchain that allows the management of decentralized identities in a context of self-sovereign identity [19]. uPort identities are fully owned and controlled by the creator, allowing centralized third party independence for creation, control or validation [6]. As shown in Figure 2.8, there are several components in the uPort identity management system [20].

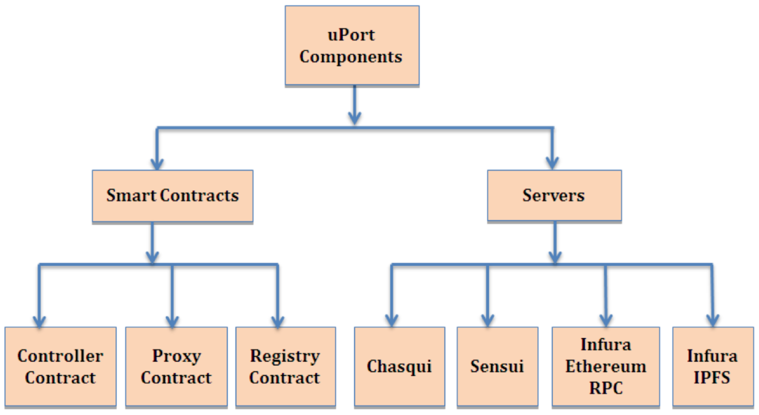


Figure 2.8: uPort components

The components of uPort can be classified into smart contracts and servers. Each of them allows the dapps (distributed applications) that integrate uPort to work with this technology. uPort abstracts the interactions with these servers and smart contracts (described below) through some libraries.

Controller Contract
It manages the overall control logic and the functionality of controlling the access to the proxy contract. It gives the user the possibility to claim their identity if the private key or the device on which the wallet app is installed is lost. It maintains also a list of recovery delegates (a list of other identities) that can help a user retrieve their uPort identity.

Proxy Contract
The Proxy contract acts as a permanent identifier for a uPort user linked with the private key of the user. It allows the user to replace their private key without affecting their permanent identity.

Registry Contract
The uPort registry is a contract which is used in the uPort system to link attributes to identities. It offers a cryptographic link between a uPort identifier and its data attributes or profile data stored off-blockchain. The IPFS (InterPlanetary File System) [17] is a peer-to-peer protocol for storing and retrieving data on a distributed file system. The registry contract uses this protocol to store identity documents about the user.

Chasqui Server
Chasqui is a server that allows all the communications between decentralized apps, the uPort mobile wallet and the other servers.

Sensui Server
Sensui is a gas fuelling Server that avoids the requirement of a new user to purchase Ether to interact with the Ethereum blockchain. It is responsible for paying the gas fees for the new user allowing them to create a new uPort account instantly.

Infura Ethereum RPC
This server consists in an API that provides a standard RPC (Remote Procedure Call). An RPC is generally used to communicate between processes on different workstations in a network. This API interface allows uPort to communicate with the Ethereum network.

Infura IPFS
This server consists in another infura API that provides a standard interface to allow uPort to communicate with the IPFS network. IPFS is distributed storage where data can be stored and accessed. In the case of uPort, the hashes of the identity documents are stored.

2.5.1 uPort mobile wallet

uPort mobile is a secure mobile self-sovereign identity wallet that allows to manage the complete control over identity and personal data.

With the uPort mobile app it is possible to:

- create an identity on the Ethereum blockchain network;
- securely log-in to applications without passwords;
- manage personal information and credentials;
- approve Ethereum transactions and digitally sign files.

This mobile wallet is the connection to the uPort platform. uPort provides open protocols for decentralized identity and interoperable messaging that enable trusted source attribution for all web communication. In particular, when the user interacts with an application smart contract, they communicate with the smart contract Controller, which forwards the transaction using the Proxy smart contract, as shown in Figure 2.9. The smart contract proxy represents a layer between the user's private key, stored on their mobile device, and the smart contract of the application they want to contact.

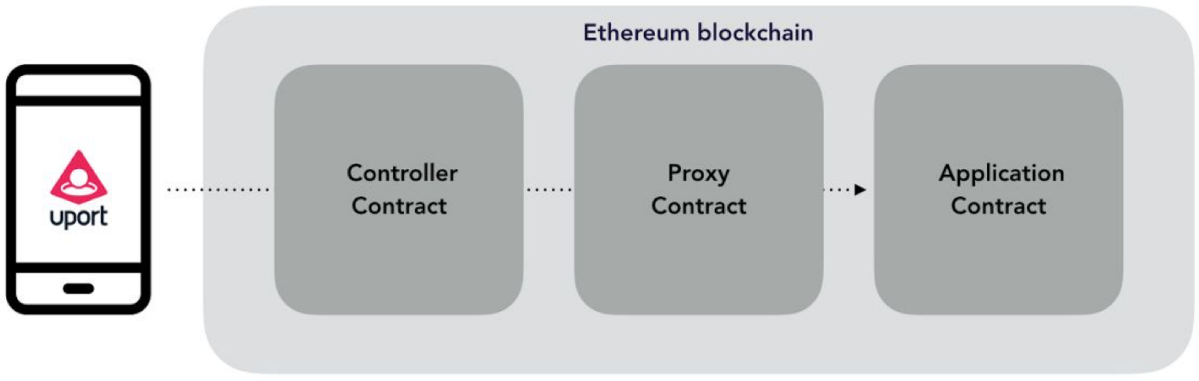


Figure 2.9: uPort app workflow

2.5.2 Libraries

uPort provides several libraries for developers to develop decentralized applications. The most important are described below.

uPort Credentials

uport-credentials is a useful library that allows developers to implement the logic that manages the verifiable credentials of the users. Through this library it is possible to:

- Create and verify authentication requests
- Request verified claims
- Verify claims about users
- Ask users to sign Ethereum transactions
- Create call requests to the Ethereum smart contract functions

uPort provides the APIs for allowing users to log into applications and share private information, such as identity information and contact details. The *uport-credentials* library is primarily used as a server-side library, where the keys used to sign the verifiable credentials can be securely stored in the filesystem. For an end-to-end solution, this library is used in combination with the *uport-transports* library (described below) in the browser to communicate between the application and user's uPort app wallet. In this case, in fact, users keep their private key stored in the uPort mobile app. Whenever the decentralized app wants to send a transaction, it presents a QR code, which is scanned by the uPort app. The uPort app signs the transaction and forwards it to the blockchain network.

uPort Transports

Another library used in the solution is *uport-transports.* This library consist of a collection of functions used to set up communication channels between an application and a uPort mobile wallet, representing an Ethereum identity. These functions consume request messages and additional transport params, then they send these strings to a uPort client. Requests to authenticate, save verifiable credentials, disclose verifiable credentials, are always made up of URLs managed by the uPort mobile app. There are three different ways to submit the URL to the uPort mobile app:

- **Open uPort URL on the phone.** On the device any URL whose scheme is *me.uport:* or hostname is *id.uport.me* will open directly in the uPort app, if installed. If the app is not installed, a website will open with a link to the App Store and Play Store. The primary purpose of these URLs is to create links in a mobile web app that contains requests to the uPort app.
- **Scan QR Code.** Encoding requests in QR codes is very intuitive on the user experience side. This modality was used in the thesis project. While this is often used to interact with an application in a desktop browser, many other applications exist. In fact, QR codes can be printed and hung on bulletin boards, displayed on projectors at conferences and so on.
- **Send Push Notification.** It is also possible to request permissions from users to send requests directly to their uPort mobile app using push notifications. On the one hand, push notifications make the interaction flow much easier for users if they have to interact with multiple requests on their phone. On the other hand, it is necessary that the user has already installed the wallet on the mobile and they may not notice the notification.

2.5.3 DID management

In the uPort DID method [27], the *Method-Specific-Identifier* was the address of a Proxy smart contract, a specific one per identity. There was also an IdentityManager contract [28] that was a controller contract for Proxy contracts that was shared between users. The idea of using an IdentityManager was to minimize gas costs compared to each user having to have their own separate controller for their proxy, while still allowing the user to have full control over their own proxy. The IdentityManager added also some interesting features often missing in SSI solutions such as the ability to revoke and rotate individual keys, and to restore access if a device is lost.

However, the uPort DID method approach required deploying one smart contract per identity, which had an associated cost that was not wanted or just unacceptable in many scenarios. Other approaches, such as the ERC725 Ethereum Identity Standard [2] faced the same exact issues. To a certain extent, identity creation should be free, and the authentication process should be feasible to operate also in an off-blockchain environment. The ethr DID method and the EIP-1056 [7] were developed to satisfy those needs.

The ethr DID method allows any Ethereum account to become a valid identity without the need for any kind of registration process. Since creating a valid Ethereum account is an off-blockchain costless process, the identity creation is then for free. It is just in the case that we need to update keys, add delegates, endpoints, that a smart contract is deployed. If an account is not listed in the contract, then it could be assumed that the key has not been changed.

In the case that key management or additional attributes are required, the deployed smart contract follows the EIP-1056 pattern, which utilizes a smart contract for a lightweight identity management system intended explicitly for off-chain usage and ease of key delegation.

EIP-1056 proposes a way of a smart contract or regular key pair delegating signing for various purposes to externally managed key pairs. This allows a smart contract to be represented, both on-chain and off-chain or in payment channels through temporary or permanent delegates.

DID creation and registration
In order to create an ethr DID, an Ethereum address (a key pair), needs to be generated. At this point, no interaction with the target Ethereum network is required. After the key pair generation, the DID has to be registered in the verifiable data registry. The DID registration is performed by a DID Registar, that is a software component that implements the DID functions. The holder of the private key is the entity identified by the DID.

DID updating
A DID Document may be updated by invoking the relevant smart contract functions as defined by the ERC-1056 standard. This includes changes to the identity owner, adding delegates and adding additional attributes, such as service endpoints. Delegates are addresses that are delegated for a specific time to perform a function on behalf of an identity. They can be accessed both on and off-chain. These functions will trigger the respective Ethereum events which are used to build the DID Document for a given identity.

DID deletion/revocation
In case no changes were written to the ERC-1056 contract, nothing special needs to be done. It is only necessary to delete the private key which belongs to the Ethereum address from the wallet on it was stored. If the ERC-1056 contract was changed, several steps should be followed:

- the owner of the smart contract needs to be set to 0x0. Although 0x0 is a valid Ethereum address, this will indicate the identity has no owner which is a common approach for invalidation also used, for instance, with tokens.

- other elements, including delegates and additional attributes, of the DID Document may be revoked explicitly by invoking the relevant smart contract functions as defined by the EIP1056. All these functions will trigger the respective Ethereum events.

2.6 Veramo framework

Veramo [11] is a JavaScript framework for verifiable data that was born as an evolution of uPort. It allows the creation of agents, which implement plugins to perform operations. Using this framework, it is necessary to implement all the expected actors in a self-sovereign identity ecosystem. Examples of operations provided by plugins include the creation of identifiers, the resolution of identifiers, the issuance of verifiable credentials, the request for disclosure and the verification of disclosure request. The Veramo plugins can be classified as shown in Figure 2.10.

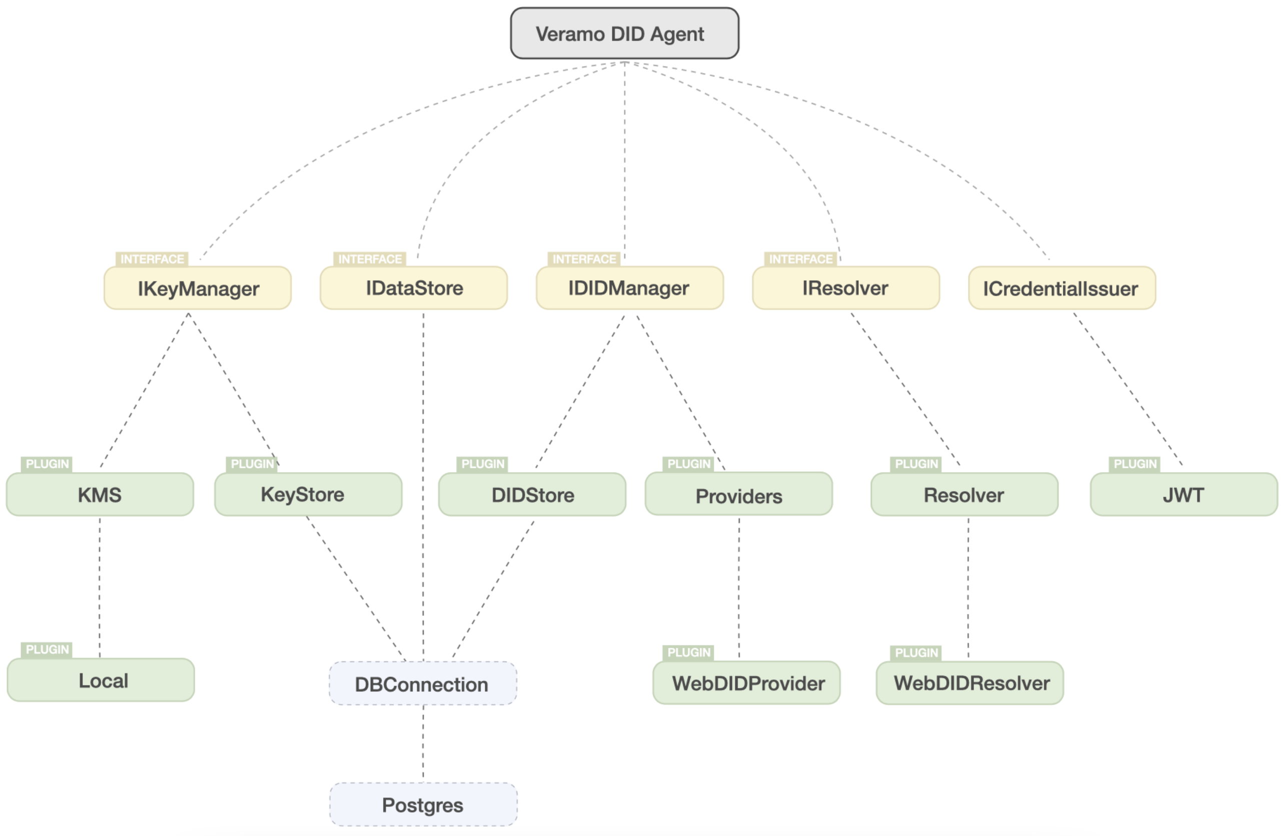


Figure 2.10: Relationship between the core agent, interfaces, plugins and external protocols, from the Veramo documentation [11]

The Veramo DID agent represents the entry point into the Veramo framework. A Veramo agent is an implementation of an agent using the plugin architecture shown in Figure 2.10, that allows it to be modular and implement only the features required for its specific purpose in the overall ecosystem. Unlike uPort, Veramo does not work through a default wallet such as the uPort mobile wallet application, so it is mandatory to implement the wallet components using an agent and giving them some capabilities, required to perform its job. For example to implement a wallet component, it is necessary to provide that agent with the plugins to create the identity, save credentials and disclosure them. To implement a verifier, on the other hand, it is necessary to use an agent with the capabilities to create disclosure requests and verify the credentials that are provided to it. To implement a verifiable credential issuer, it is necessary to use an agent with the capabilities to generate verifiable credentials and sign them.

3 Case study

This chapter presents an overview of the i3-Market project on which this thesis is based. Section 3.1 describes the project from a high-level perspective. Section 3.2 represents different use cases of the marketplace. Section 3.3 presents the overall architecture of i3-Market with a particular focus on the Trust, Security and Privacy package. Section 3.4 describes the contribution of this thesis to the project.

3.1 i3-Market

The i3-Market project [13] is a H2020 project that addresses the growing demand for a single European data market economy. Nowadays, despite various research and innovation attempts working on big data management, there is no broadly accepted trusted and secure data marketplace at European level. The i3-Market project will address this by developing lacking technologies and solutions for a trusted, interoperable and decentralised infrastructure, called i3-Market Framework, that enables federation via interoperability of the existing and future emerging data spaces and marketplaces. The main objective of i3-Market is therefore to develop the building blocks for data providers and consumers, and thus incentivise and enable the creation of a more trusted European data market economy. In fact, the growth of the data economy is possible by innovating market platforms and demonstrating it with industrial implementations. The i3-Market project addresses the interoperability and integration challenge for trading data assets across independent stakeholders. This can be done by means of secured transactions based on semantics data annotation and a trusted data trading platform that will provide a network of decentralised and economy-driven and scalable data repositories. This platform can be extended for enabling the deployment of intelligent industrial data services fostering innovation and business opportunities.

3.2 Use case scenarios

A marketplace is a brokerage website for the purchase and sale of a good or service. In the i3-Market architecture, several websites that are part of the consortium are present. There are two types of marketplace users: data provider and data consumer.

A data provider is an entity, organization, company that produces data. These data can be collected for example through IoT devices and are called "operational data". In order to make them usable, they must be processed, normalized and cleaned, becoming metadata.

A data consumer is an individual, group, or application that receives data in the form of a collection. They use the data for queries, market analysis, reports and so on.

Each website represents a data provider, that uploads data in one of the websites and makes them available on the marketplace. These data can be purchased or rented through services by data consumers. All of these websites share APIs to interact with the blockchain network and smart contracts that manage the business logic of the framework.

Some of the most common activities that users can do in the marketplace are:

- Generate data (data provider)
- Register a new offering (data provider)
- Purchase data (data consumer)

The activities that users can do within the i3-Market marketplace can be described with the following use case scenarios.

Scenario 1: Generate data
The generation of data takes place at the sites of the data owner. They can come from different sources such as a car IOT device, from a manufacturing plant, or from a person producing data. These data are stored in a data space or a data storage.

Scenario 2: Register a new offering
New data offerings are new sets of data that are offered for exchange within the marketplace. For instance, it could be a new machine that is sending data to the data space. The new data offering has to be described with metadata. In order to simplify this task and to ensure that the descriptions are created in a uniform way, semantic templates are used by data provider.

Scenario 3: Purchase data
Data consumers are users that need data and visit the marketplace for that. On the marketplace they will search for data by defining metadata that describes their needs. The i3-Market backplane system compares the search request with all registered offerings and creates a list of search results. This list is transferred to the marketplace. The data consumer can filter the data (e.g. by region of origin) and view details about the search results. If they find some interesting data sets, they may decide to buy it. In that case they will review the payment info (e.g. price, payment method) and create a data request. There will be a smart contract template that can be used for the legal aspects such as the notarization of the purchase. Following the notarization, a verifiable credential is issued to the user, in order to update his profile by adding a certificate that guarantees his purchase. In this thesis we focus on aspects from this scenario.

3.2.1 Project pilots

The consortium will implement and integrate the i3-Market framework in three industrial domains with three industrial pilots. Each pilot project aims to examine the viability of the implementation, test the reliability and validate whether it is scalable in real industrial environments. The project pilots are:

- Automotive Sector: this scenario places itself between original equipment manufacturers and service providers through a common vehicle information model, removes all brand-dependent data formats, thus unlocking the potential of vehicle data.
- Intelligent Manufacturing Sector: in this pilot, the key is the protection of sensitive industrial information. Therefore, the i3-Market solution should explicitly specify who can access operational data and also to revoke this approval at any time.
- Healthcare and Wellbeing Sector: this pilot leverages IoT technologies to address patients and relatives needs while protecting autonomy, privacy, safety, and focuses on enabling well-being in a cost-efficient manner.

Trying to make the project's applications real and taking as a reference the pilot project related to the automotive sector, the use cases could be the following:

- Data generation: various car manufacturers produce cars with IOT devices capable of generating data on journeys, average speeds, driving time and driving style. This data is processed in order to remove specific information about the car model, the manufacturer and so on.

- Offer registration in the marketplace: the aggregated data builds one or more datasets that can represent, for example, the busiest road sections, sections where the average speed is greater or too high, and so on.
- Data purchasing: an advertising company may be interested in buying or renting the dataset to find out which are the busiest stretches of road to install an advertising billboard, or another company may be interested in knowing the stretches of road where the speeds are above average to install speed cameras.

3.3 i3-Market framework architecture

Building a framework that allows many websites to connect and cooperate as a single marketplace requires several components. Many aspects must be taken into consideration, such as where and how to save data, according to which formats and templates, how to be able to query large amounts of data, how to access data effectively, and so on. Figure 3.1 describes at a high level what the main components are and how they relate to each other.

Afterwards, I will focus on the Trust, Security and Privacy System package, in order to locate the placement of the work of this thesis in the context of the project. This package contains all the features to ensure security in such a large system. In particular, the model used for user management is based on self-sovereign identity. Self-sovereign identity is a decentralized digital identity model based on blockchain and distributed ledger technologies. This model allows users to own their data. This means that all user attributes, such as identity information, purchases, tokens available in their wallet and so on are maintained by the user in a decentralized manner. To manage all this, we have designed specific submodules for the various tasks. The submodules contain services that are used to manage operations such as regulate access and permissions on the Data Storage System and access to the APIs of the Backplane System, information disclosure, notarization of blockchain operations, and non-repudiation protocols.

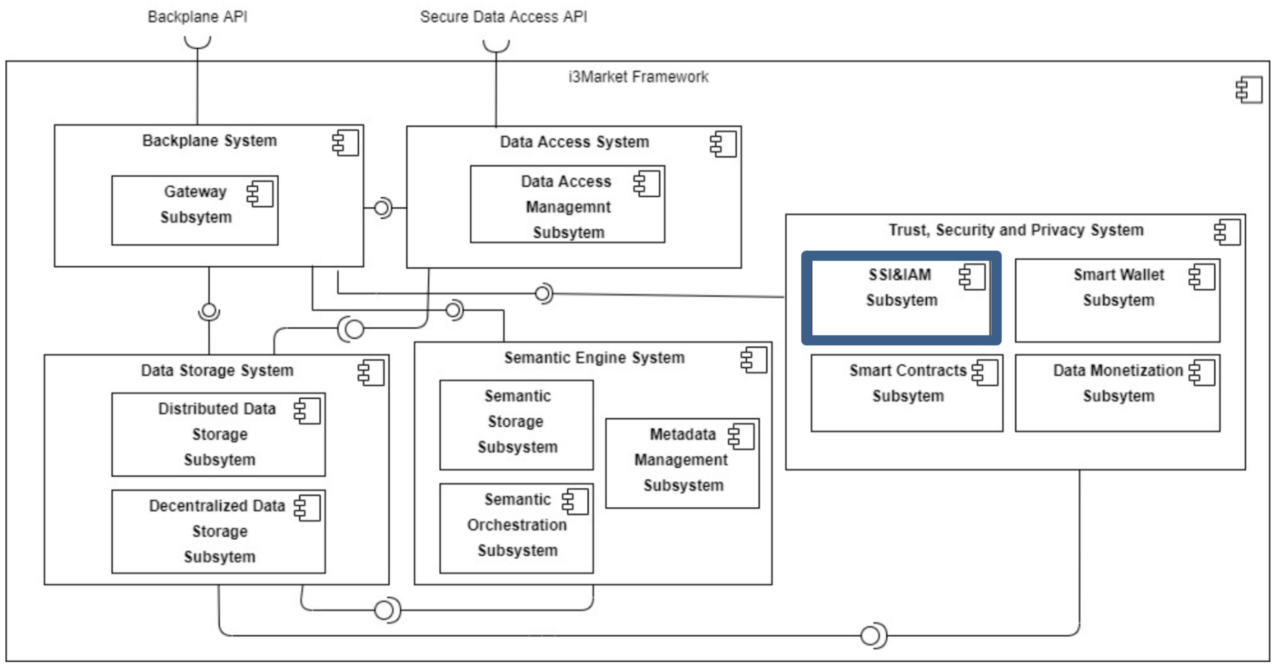

Backplane API
Secure Data Access API
i3Market Framework
Backplane System
Gateway Subsytem
Data Access System
Data Access Managemnt Subsytem
Trust, Security and Privacy System
SSI&IAM Subsytem
Smart Wallet Subsytem
Smart Contracts Subsytem
Data Monetization Subsytem
Data Storage System
Distributed Data Storage Subsytem
Decentralized Data Storage Subsytem
Semantic Engine System
Semantic Storage Subsystem
Semantic Orchestration Subsystem
Metadata Management Subsystem

Figure 3.1: i3-Market Framework overview

In particular, the submodules of the Trust, Security and Privacy System module are the following:

- Self-Sovereign Identity & Identity and Access Management subsystem (SSI&IAM), which manages authentication, authorization and verifiable credentials.
- Smart Contract subsystem, which manages smart contracts and interfaces with the blockchain. The smart contracts subsystem consists of a service that allows the deployment and interaction with smart contracts. It interfaces with the i3-Market blockchain, which is the open-source Ethereum client Hyperledger Besu [10].
- Smart Wallet subsystem, which includes a wallet to save user information. The smart wallet subsystem will consists of a set of wallets. In particular, there will be an app wallet and a web-based software wallet. In the first release of the project and therefore also in this thesis, uPort is used as the only wallet.
- Data Monetization subsystem, which settles payments and token exchange in a trustworthy manner. The federation of independent data spaces/marketplaces further calls for a highly secure, trusted and cost-efficient payment solution. Therefore, it is needed a crypto currency solution that allows instant currency exchange among the participating data spaces/marketplaces, and also support full audibility of all transactions. This is vital for a fully decentralised solution, as it provides the basis for building trust in the whole system.

3.3.1 Identity and access management subsystem (SSI&IAM)

Since the content described in this thesis is related to a service belonging to this submodule, I am going into more detail. The Identity and Access Management submodule consists of two components:

- an OAuth 2.0 Authorization Server with OpenID Connect that allows users to authenticate disclosing verifiable credentials;
- a microservice to issue, revoke and verify verifiable credentials following the W3C recommendation.

The access to i3-Market resources and services is guaranteed by the OAuth 2.0 Authorization Server through the authorization code flow with PKCE. Nowadays this is the recommended flow for native applications. The idea behind the design is to make the IAM subsystem to be the only one that needs to verify DIDs and verifiable claims and let the rest of the components and stakeholders stick to standard implementations of OIDC-OAuth2. In this way, it will be easy to use the i3-Market identity provider as a means of authenticating the different i3-Market stakeholders in any protected API such as backplane API, marketplaces API, providers API. The authentication is performed with DIDs and, during the consent phase of the authentication, users are asked to prove claims using verifiable credentials.

The second component of the submodule is the microservice for issuing, verifying and revoking verifiable credentials. It allows users to obtain verifiable credentials that certify their purchases in the marketplace and build their digital identity. In this way, users hold attestations that can be revealed during authentications and authorizations. These credentials are held exclusively by users in their wallet.

3.4 Contribution in the project

The goal of this thesis is to design and implement a SSI&IAM module, which is responsible for managing verifiable credentials compliant with the W3C standard. In particular, it describes the microservice I implemented for issuing, verifying and revoking verifiable credentials in a self-sovereign identity context and how these features are used in the i3-Market project. As part of the project, this microservice interfaces with other modules such as the one for authentication via OIDC, the module for managing smart contracts and the software wallet module. In fact, authentication and authorization in the context of self-sovereign identity uses certified attributes maintained by the user in their wallet in the form of verifiable credentials. The design and the implementation of the solution for the microservice to generate verifiable credentials is described in Chapter 4. The integrations with the Open ID Connect provider and the i3-Market software wallet are described in Chapter 5.

4 Solution design

This chapter presents the implementation details of the module I developed to manage verifiable credentials within the i3-Market project. This module is a component of the Trust, Security and Privacy System package presented in Figure 3.1. In order to manage verifiable credentials, I implemented a Node.js microservice. This microservice provides some APIs that implement the core functions to manage verifiable credentials, namely issuing, revoking and verifying verifiable credentials, and a utility function. Section 4.1 presents from a technical point of view the workflow of each operation, taking as a reference the actions that are performed within the marketplace. Section 4.2 provides some more details on the strategy adopted to document and deploy the microservice.

4.1 Verifiable credentials microservice

Verifiable credentials are useful in a self-sovereign identity ecosystem as they assert information about the user to whom a credential is issued. In the context of the i3-Market project, users are asked to disclose verifiable credentials, which attest particular attributes about a purchase in the marketplace. The certified attributes and permissions are used to obtain an OAuth access token and an ID token that allow the use of services or access to resources in the marketplace. Verifiable credentials are therefore used as proof method in the authorization flow as described in Section 5.1. If a credential is valid, it means that the user is authorized to access a resource or service that requires the holding of that credential. For this reason, a service that generates verifiable credentials is necessary. Once a verifiable credential is saved by users in their wallets, anyone who receives the verifiable credential and has access to the DID of the users can then confirm that the verifiable credential has been issued by a trusted entity and has not been revoked for some reason.

As shown in Figure 4.1, there are multiple instances of data providers and an instance of data marketplace. Each instance is deployed in a separated server and has its own OIDC modules and its own Verifiable Credential microservice to generate, verify and revoke verifiable credentials. In relation to the roles of the W3C Recommendations on verifiable credentials [29]:

- the OIDC module plays the role of the verifier
- the Verifiable Credential microservice is the issuer, with an extra feature, i.e., the one to retrieve the DID of the user
- the i3-Market user is the identity holder, i.e., the one who holds his identity and verifiable credentials in his wallet.

Each instance of the Verifiable Credential microservice has its own DID and private key used to sign verifiable credentials. For this reason, each verifiable credential has as its issuer the DID of the microservice that generated it. In this way it is possible to know exactly if the entity that issued the credential and therefore it is possible to know if it is certified and trusted in the system. Likewise, for revocation, only the microservice that generated a credential has the privilege to revoke it. This prevents unauthorized revocation of verifiable credentials.

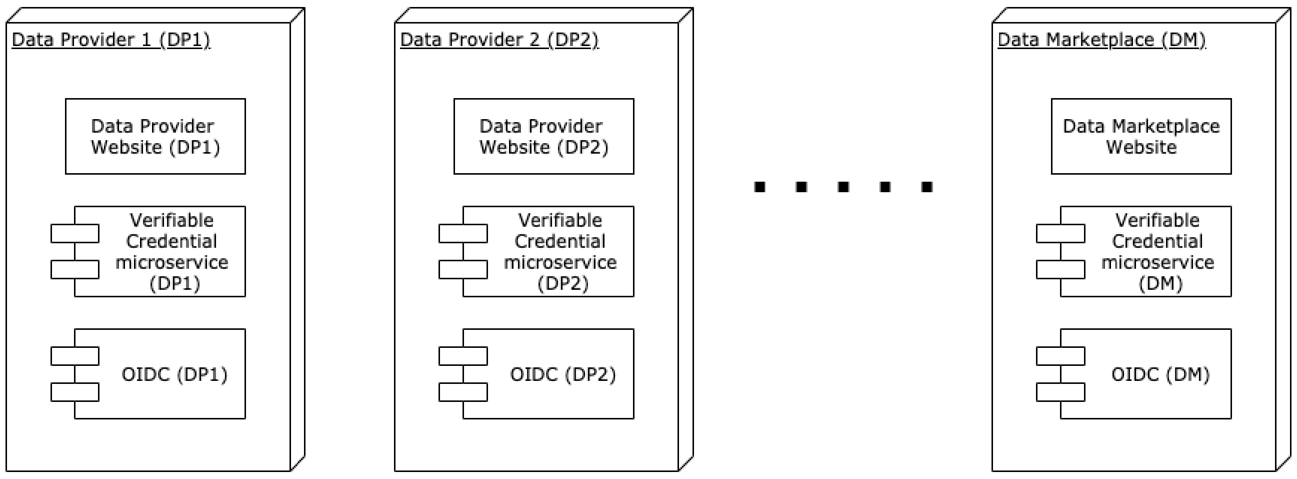

Data Provider 1 (DP1)
Data Provider Website (DP1)
Verifiable Credential microservice (DP1)
OIDC (DP1)
Data Provider 2 (DP2)
Data Provider Website (DP2)
Verifiable Credential microservice (DP2)
OIDC (DP2)
Data Marketplace (DM)
Data Marketplace Website
Verifiable Credential microservice (DM)
OIDC (DM)

Figure 4.1: i3-Market service instances

The Verifiable Credential microservice provides APIs that implement the main functions necessary for the management of verifiable credentials. The functionalities are the following:

- Issue of a verifiable credentials. This is done by displaying a QR code that the uPort wallet application scans to store the credential.
- Revoke a verifiable credential. This is done by writing the hash of the verifiable credential on a smart contract. This smart contract acts as a registry, in which each record on this registry represents a specific verifiable credential.
- Verify a verifiable credential. This is done by checking that a credential is valid, not expired and has not been marked as revoked in the smart contract revocation registry.
- Get a user's DID. In this implementation the uPort technology was used, which through its framework allows to release a verifiable credential associated with a DID. To obtain a user DID, this feature is present which, by scanning a QR code, is able to receive the DID of the user who is scanning it.

The implemented workflows related to the APIs are presented in the following subsections, taking as reference the use cases for i3-Market. In particular, the APIs of the Verifiable Credential microservice are used in some flows such as the registration (Section 4.1.1), the issuance of verifiable credentials in relation to a purchase (Section 4.1.2), the revocation of the verifiable credential if, for example, the terms of use of a service within the marketplace are not respected (Section 4.1.3), and finally the verification of a verifiable credential by a verifier (Section 4.1.4).

4.1.1 Registration flow

The first thing a user does in the marketplace is the registration. The goal of the registration process is to issue a verifiable credential that certifies the type of a user, which can be a data consumer or a data provider. The workflow for the registration process is described in Figure 4.2. The entities involved are the following:

- the Identity Holder, that is the i3-Market user
- the User Agent, that is also the Client of the OIDC (i3-Market Data Marketplace website)
- the uPort application, which is the wallet in which credentials are stored
- the Verifiable Credential microservice (i3-Market Data Marketplace instance)
- the i3-Market Data Marketplace backend

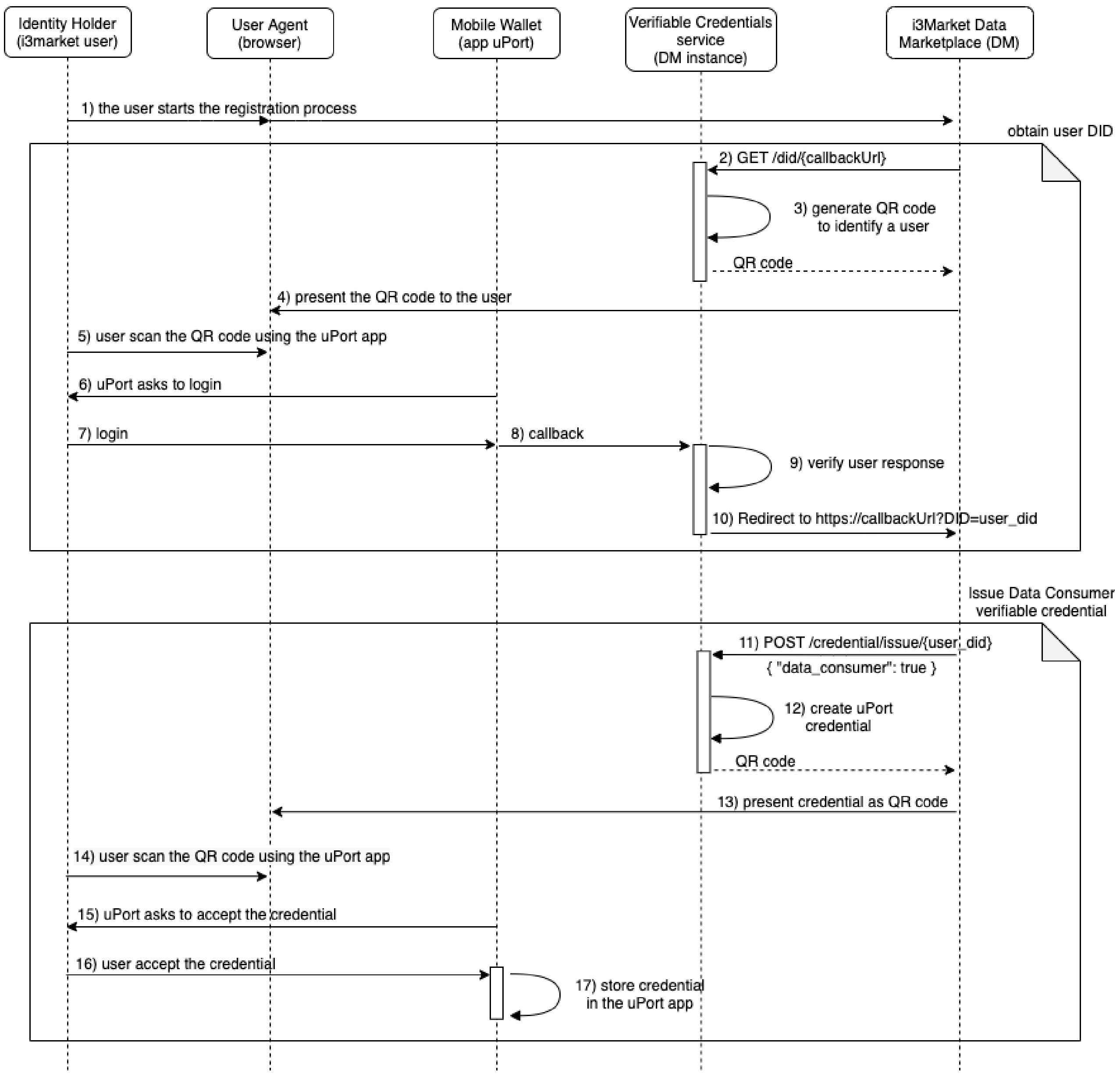

Identity Holder (i3market user)
User Agent (browser)
Mobile Wallet (app uPort)
Verifiable Credentials service (DM instance)
i3Market Data Marketplace (DM)
1) the user starts the registration process
obtain user DID
2) GET /did/{callbackUrl}
3) generate QR code to identify a user
QR code
4) present the QR code to the user
5) user scan the QR code using the uPort app
6) uPort asks to login
7) login
8) callback
9) verify user response
10) Redirect to https://callbackUrl?DID=user_did
Issue Data Consumer verifiable credential
11) POST /credential/issue/{user_did}
{ "data_consumer": true }
12) create uPort credential
QR code
13) present credential as QR code
14) user scan the QR code using the uPort app
15) uPort asks to accept the credential
16) user accept the credential
17) store credential in the uPort app

Figure 4.2: User registration workflow

At the beginning, the user enters his registration data (1), relating to a data consumer user or a data provider. When a user is registered as a data consumer or data provider, he is registered for all the sites of the i3-Market consortium. For this reason, this type of verifiable credential is issued exclusively by the i3-Market Data Marketplace entity. The user's data are entered on registration forms in a dedicated section of the i3-Market Data Marketplace. In order to know for which DID the issuance of the verifiable credential should be requested, the i3-Market Data Marketplace must obtain the DID of the user. This part of the flow involves the Authenticate API, which is specifically designed to retrieve the DID of a user who already has a wallet. In particular, the i3-Market Data Marketplace performs a GET to the authenticate API of the Verifiable Credentials microservice (2) passing a *callbackUrl* parameter, which indicates the URL where to send the DID as a response. When the i3-Market Data Marketplace initiates this API call, the Verifiable Credentials microservice performs a *createDisclosureRequest* (3), through the uport-credential library. This operation resolves in the following JWT:

```
{
  "iat": 1619882317,
  "exp": 1619882917,
  "permissions": [
    "notifications"
  ],
  "callback":
      "https://6b3c7a1d5fe3.ngrok.io/did/callback/0242ac130002/datamarketplace.i3market.eu",
  "type": "shareReq",
  "iss": "did:ethr:0x9b614f69adcc7f533565bd760134c8db3180c3a6"
}
```

The fields of the JWT represent the following information:

- iat: issued at (seconds since Unix epoch)
- exp: expiration time (seconds since Unix epoch)
- permission: array of permissions requested. The only supported is notifications
- callback: callback url necessary in step (8)
- type: that indicates a Selective Disclosure Request operation
- iss: issuer (who created and signed this token, i.e., the verifiable credential microservice)

This JWT is finally encoded as a QR code via the uport-transport library and sent back to the i3-Market Data Marketplace as a response from the API. Then, the QR code is shown to the user on an HTML page of i3-Market Data Marketplace (4), as presented in Figure 4.3.

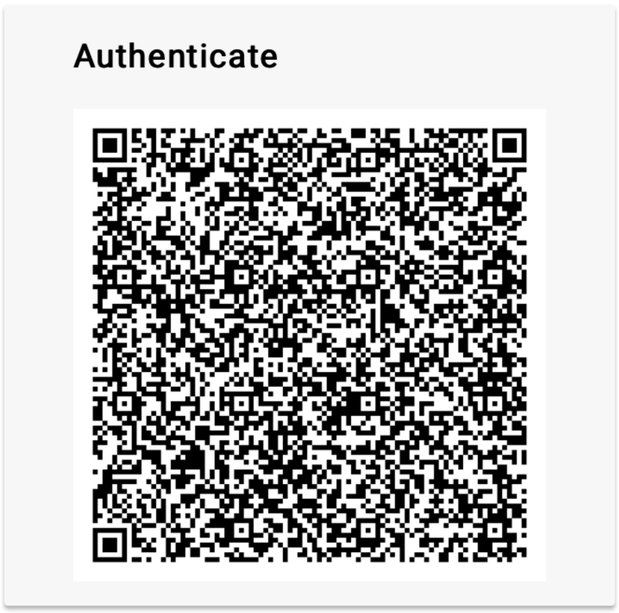
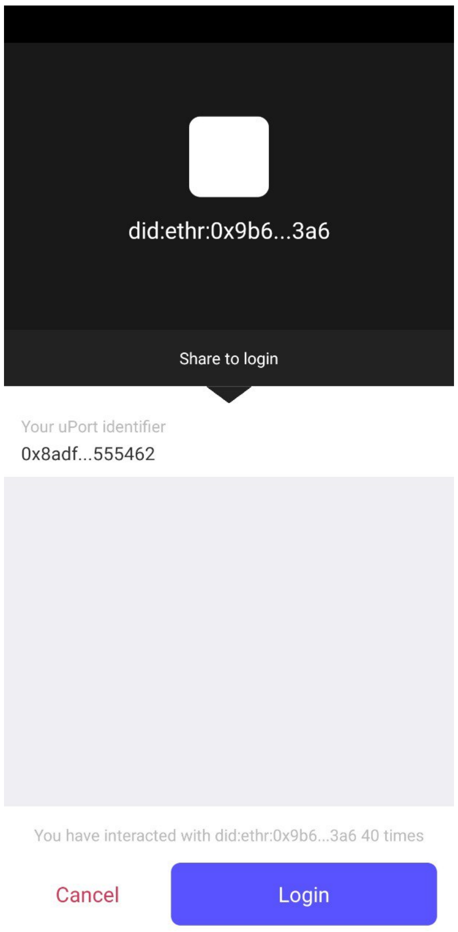


Figure 4.3: QR code to authenticate a user

At this point the user, through his uPort app, scans the QR code (5). The uPort app will recognize the QR code and ask the user if they want to log in (6), as shown in the Figure 4.4.



Your uPort identifier
0x8adf...555462

You have interacted with did:ethr:0x9b6...3a6 40 times

Figure 4.4: uPort app authentication request

If the user agrees to login (7), the uPort app sends the following access token to the Verifiable Credentials microservice (8) via callback.

```
{
  "iat": 1619883621,
  "exp": 1619970021,
  "aud": "did:ethr:0x9b614f69adcc7f533565bd760134c8db3180c3a6",
  "type": "shareResp",
  "own": {},
  "req": "eyJ0eXAiOiJKVk ... ji5RPvgBs1QE",
  "capabilities": [
    "eyJ0eXAiOiJKV1 .... wxaYFQE"
  ],
  "boxPub": "HoXkkUmtBiN9Jn9G3QQTPPPbW31CDHCjDUJcn2KgHB8=",
  "iss": "did:ethr:0x8adfb8543c3c14b83c3a104ed8aeb19b64555462"
}
```

The fields of the JWT represent the following information:

- aud: audience, i.e., who or what the token is intended for (the Verifiable Credential microservice)
- type: that indicates a Selective Disclosure Response operation
- own: the signed claims requested from a user. In case of authentication, no claim is required as the purpose is to obtain the DID
- req: the original JWT encoded Selective Disclosure Request
- capabilities: an array of JWT tokens giving client app (i.e. the Verifiable Credential microservice) the permissions to send push notifications to the uPort app
- boxPub: 32 byte base64 encoded Curve25519 public key of requesting identity. Use to encrypt messages sent to callback URL
- iss: who created and signed this token, i.e., the user using the uPort app

At this point the Verifiable Credentials microservice performs an *authenticateDisclosureResponse* (9), giving in input the previous JWT. This operation is also provided by the uport-credential library. If the *authenticateDisclosureResponse* operation is successful, the following JWT is generated.

```
{
  "did": "did:ethr:0x8adfb8543c3c14b83c3a104ed8aeb19b64555462",
  "boxPub": "HoXkkUmtBiN9Jn9G3QQTPPPbW31CDHCjDUJcn2KgHB8=",
  "pushToken": "eyJ0eXAiOiJKV1Qi ... xK9CDc6C3FpaAMFmwxaYFQE"
}
```

The pushToken is itself a JWT token which contains the following information:

```
{
  "iat": 1619883621,
  "exp": 1651419621,
  "aud": "did:ethr:0x9b614f69adcc7f533565bd760134c8db3180c3a6",
  "type": "notifications",
  "value":
      "arn:aws:sns:us-west-2:113196216558:endpoint/GCM/uPort/da1d6ac4-35fa-3c22-9190-09436ed60216",
  "iss": "did:ethr:0x8adfb8543c3c14b83c3a104ed8aeb19b64555462"
}
```

From the previous JWT it is possible to extract the DID of the user who has logged in. Then, a redirect is performed to the callbackUrl passing the DID as a query parameter (10). It will be the task of the i3-Market Data Marketplace to manage this callback route in order to intercept and save the DID.

The second part of the flow relates to the issue of a verifiable credential to certify the role of the user (e.g., a data consumer). At a high level, issuing a uPort credential involves two steps:

- Cryptographically signing user data on behalf of a uPort server application, i.e., the Verifiable Credentials microservice
- Send uPort credentials as a JWT to users via a QR code, push notification, or another transport mechanism to the uPort app. In this implementation, the uport-transport library was used to send credentials to the user's uPort app.

In order to create a verifiable credential, the i3-Market Data Marketplace performs an API call to the Verifiable Credentials service at the POST /credential/issue/{did} endpoint (11), communicating the user's DID just retrieved and the credential as form-data in the following format:

```
{
  "data_consumer": true
}
```

The i3-Market Data Marketplace may request the issuance of credentials only relating to the registration process, i.e., *data_consumer* and *data_provider*. All other credentials, relating to the purchase of assets or services, can be requested by Data Providers, as described in the Section 4.1.2. When the API is called, the Verifiable Credential microservice performs the uPort *createVerification* function (12), provided by the uport-credential library, that resolves to a JWT like the following:

```
1 {
2   "iat": 1615836186,
3   "exp": 1618428186,
4   "sub": "did:ethr:0x8adfb8543c3c14b83c3a104ed8aeb19b64555462",
5   "claim": {
6     "data_consumer": true
7   },
8   "iss": "did:ethr:0x9b614f69adcc7f533565bd760134c8db3180c3a6"
9 }
```

In this JWT the field *claim*, in row 6, contains the information of the credential to be generated. In the case of the registration process, the information relates to the type of user, i.e. that the user is a data consumer. The verifiable credential is a JSON object and as an implementation choice it was decided to create a credentials in the format *"string":boolean*. However, it is possible to generate more structured credentials, containing more user information such as name, surname, email, social security number, date of birth, and so on. The fields of the previous JSON represent the following information:

- exp: the expiration time of claim, that identifies the expiration time on or after which the JWT must not be accepted
- sub: subject (whom the token refers to, i.e., the user)
- claim: the claim of the credential in object format
- iss: issuer (who created and signed this token, i.e., the Verifiable Credential microservice)

The QR code is then embedded into an HTML card and returned as API response to the i3-Market Data Marketplace. The Data Marketplace presents the QR code to the user (13), displaying it in an HTML page, as presented in Figure 4.5:

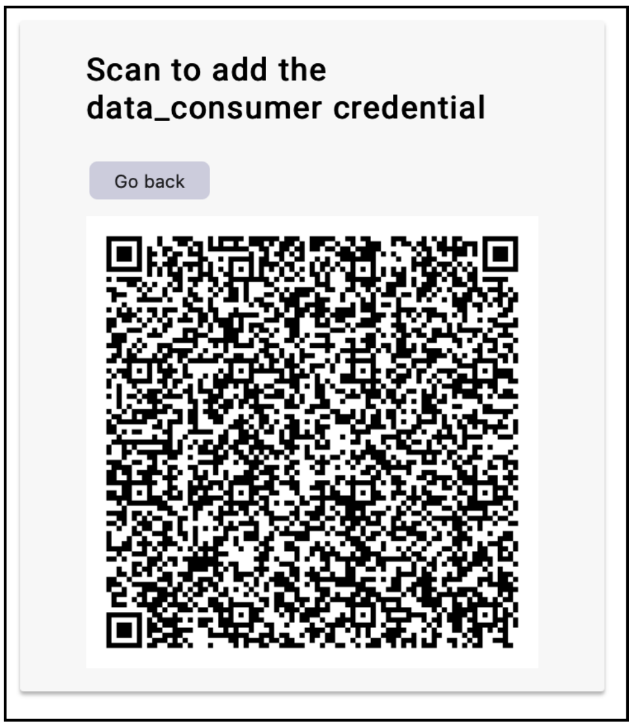


Figure 4.5: Issuing credential page

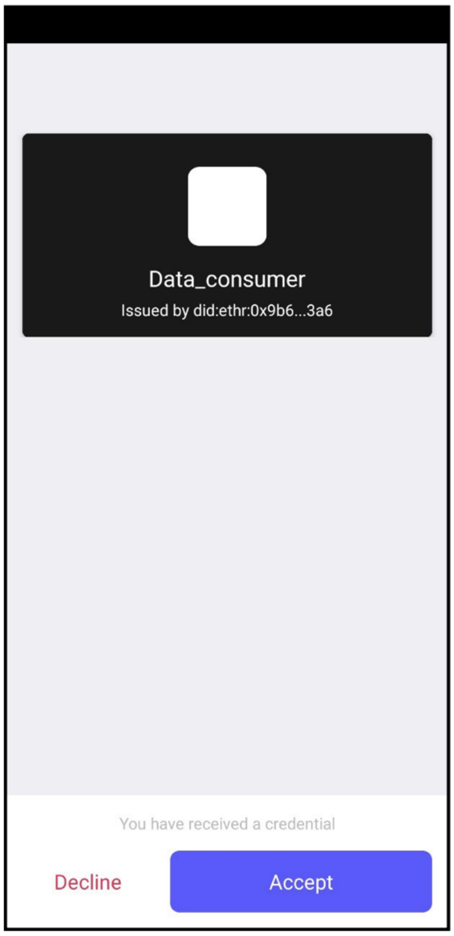
The user can scan the QR code through the uPort app (14). At this point, the verifiable credential already contains a subject, that is the DID of the user to whom this verifiable credential is intended. If the QR code is scanned by another uPort app, i.e. by another user with a different DID, the credential will not be saved in the wallet. In fact, uPort provides a check on the recipient's DID, and saves in the wallet only verifiable credentials that are actually intended for the DID registered on the app. After the scan, a push notification will appear in the uPort mobile app, containing the verification in the following JWT format:

```
{
  "header": {
    "typ": "JWT",
    "alg": "ES256K-R"
  },
  "payload": {
    "iat": 1615836286,
    "sub": "did:ethr:0x8adfb8543c3c14b83c3a104ed8aeb19b64555462",
    "claim": {
      "data_consumer": true
    },
    "exp": 1618428286,
    "iss": "did:ethr:0x9b614f69adcc7f533565bd760134c8db3180c3a6"
  },
  "signature": "Fm3_0Bh .... ML24ursVl_dZ4ikQSTgA",
  "data": "eyJ0eXAi .... A0ZSJ9"
}
```

In the payload of this JWT we can notice two new parameters:

- signature: that certifies that only the party holding the private key is the one that signed it.
- data: the JWT payload encoded

When the uPort app receives this JWT, it verifies its signature. Each Signed Message has an *iss* attribute that contains a DID of the issuer. To resolve the public key of the message, a did-resolver is used. uPort currently supports many DID methods such as *ethr* (based on ERC-1056), *uport* (uport for legacy uPort identities) and *muport* (for an experimental did resolver using IPFS and Ethereum). Being Hyperledger Besu the reference blockchain, the users' DID is *ethr*. After the signature verification, the uPort app asks the user to accept the credential (15). When the user accepts the credential (16), it will be saved in his wallet and then be present and visible in the credentials tab, that contains the list of the credentials registered in the app, as shown in Figure 4.6.


Data_consumer
Issued by did:ethr:0x9b6...3a6
You have received a credential
Decline
Accept

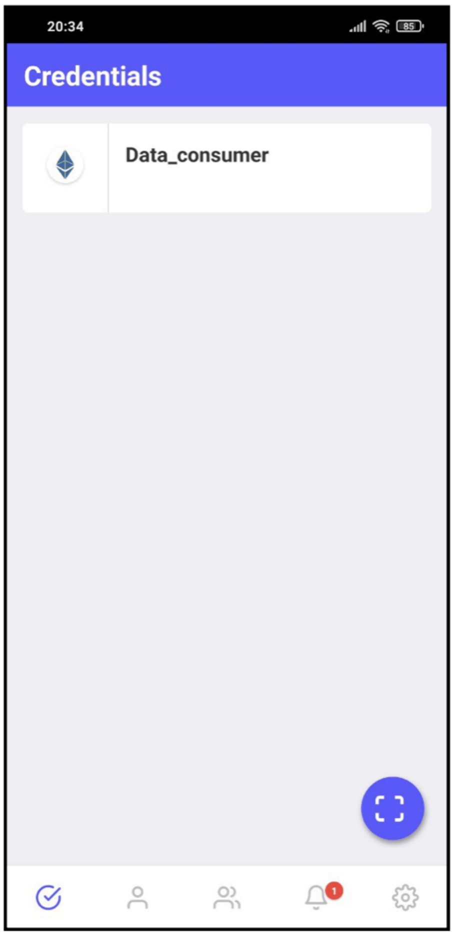


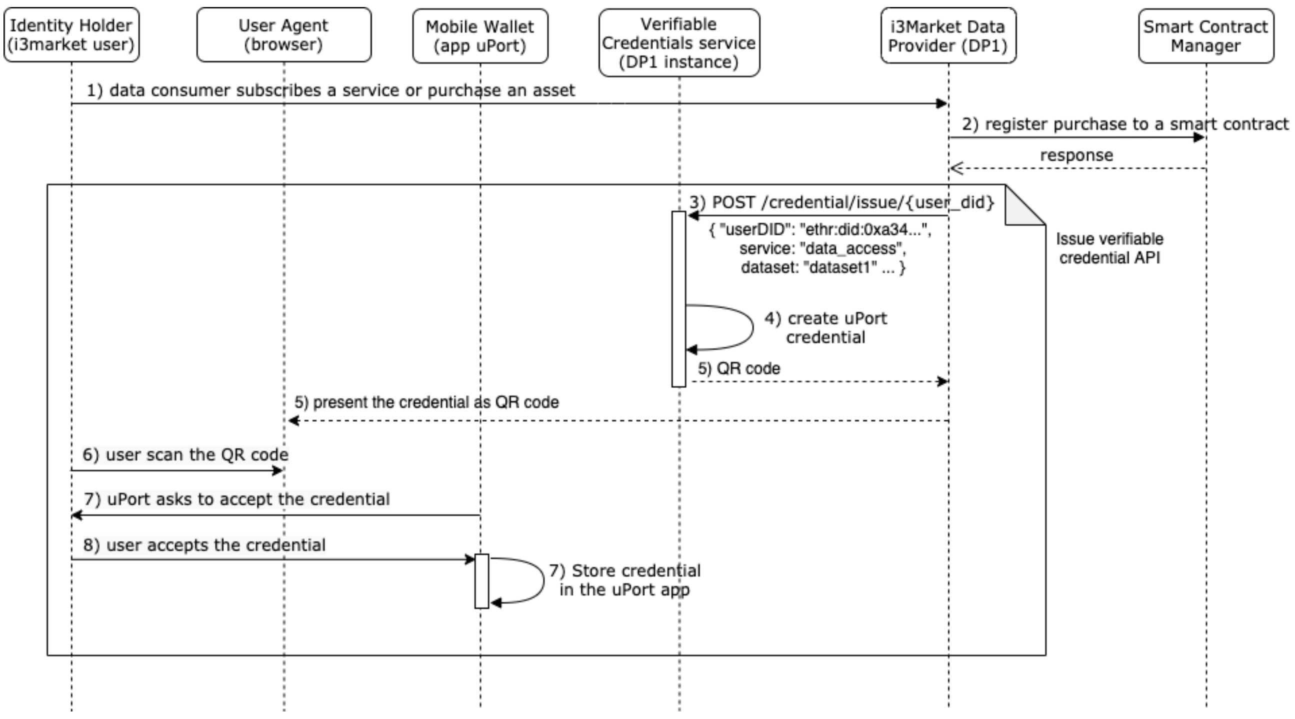
Figure 4.6: Accept credential notification in the left, list of uPort credentials in the right

This credential can be disclosed in the login process with the i3-Market Data Marketplace, as proof that the user is a data consumer or a data provider.

4.1.2 Purchase flow

Another scenario in which a verifiable credential can be issued is related to the purchase of an asset or the subscription to a service. The process of issuing a verifiable credential in the purchase scenario is shown in Figure 4.7. Compared to the previous flow, there are two different entities:

- the Smart Contract Manager, a module of the Trust, Security and Privacy System package, which deals with communicating with smart contracts
- the i3-Market Data Provider backend



Identity Holder (i3market user)
User Agent (browser)
Mobile Wallet (app uPort)
Verifiable Credentials service (DP1 instance)
i3Market Data Provider (DP1)
Smart Contract Manager
1) data consumer subscribes a service or purchase an asset
2) register purchase to a smart contract
response
3) POST /credential/issue/{user_did}
{ "userDID": "ethr:did:0xa34...", service: "data_access", dataset: "dataset1" ... }
Issue verifiable credential API
4) create uPort credential
5) QR code
5) present the credential as QR code
6) user scan the QR code
7) uPort asks to accept the credential
8) user accepts the credential
7) Store credential in the uPort app

Figure 4.7: Issue credential workflow

In order to issue a verifiable credential, the user is already registered in the marketplace. After the login via the OIDC module on a data provider website, a data consumer user can purchase assets or subscribe to services (1). At each purchase or subscription, the i3-Market Data Provider of that asset or service asks to the Smart Contract Manager to notarize the operation, through a transaction on a smart contract (2). Notarization on smart contracts ensures that the transaction is recorded with the necessary information, concerning for example the user, the timestamp, the amount of tokens spent, and so on. After that, the Data Provider asks to its instance of the Verifiable Credential microservice to issue a verifiable credential to certify that particular purchase (3). The steps from (4) onwards are the same as described in Section 4.1.1, about to the issue of the credential relating to the registration process. The difference among these credentials lies in the JWT claims, which assert different information about the users depending on the purchases they have done in the marketplace. At the end of the flow, the user obtains the verifiable credential that certifies the purchase or subscription. The verifiable credential is saved in the uPort app wallet. Once a user has saved some credentials in his wallet, he can disclose them in authentication requests, in order to certify that he holds the credentials needed to access resources or services. This part of the flow is detailed in Section 5.1, which deals with integrating the OIDC identity provider with verifiable credentials.

4.1.3 Revoke a credential

As part of the process of working with verifiable credentials, it is not only necessary to issue credentials, but sometimes it is also necessary to revoke them. The ability to revoke a credential when it is no longer valid is a core function in a verifiable credential ecosystem. For example, suppose an i3-Market data provider issues a credential to access a service, and a data consumer violates the terms of use. The data provider determines that the user has violated the terms of use and, consequently, wants to suspend the access to the service. In this way, the status of the verifiable credential needs to be changed and the next time a relying party checks the status, they will be able to see that the user is no longer valid and consequently not authorized to access the service. In order to satisfy this requirement, an API to revoke credentials has been implemented and the workflow to revoke a credential is described in Figure 4.8.

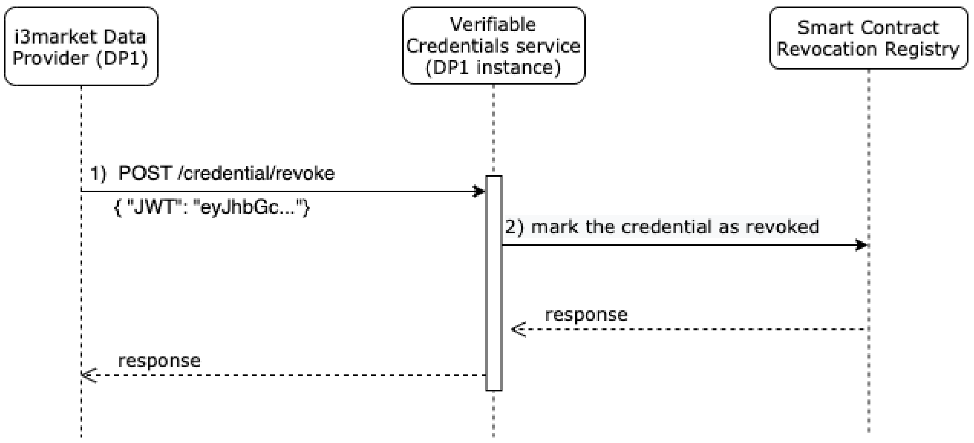

i3market Data Provider (DP1)
Verifiable Credentials service (DP1 instance)
Smart Contract Revocation Registry
1) POST /credential/revoke
{ "JWT": "eyJhbGc..."}
2) mark the credential as revoked
response
response

Figure 4.8: Revoke credential workflow

At the beginning of the flow of Figure 4.8, the data provider calls the API of the Verifiable Credentials microservice (1) communicating that a specific credential belonging to a user must be marked as revoked. The verifiable credential to be revoked is passed through the body parameter in the form { *"JWT": "eyJhbGc ..."*}. As an implementation choice, it was decided that only those who issued a credential are allowed to revoke it. To satisfy this requirement, a check is made, if the issuer of the credential is the address of the issuer then it proceeds, otherwise it immediately blocks the flow. When the verifiable credentials microservice receives the API call, it writes the credential hash through a transaction in a smart contract named RevocationRegistry (2), in order to keep track of the action performed. Since the i3-Market blockchain is an Ethereum-type blockchain, the smart contract is written in solidity and its code is the following:

```
1  pragma solidity ^0.5.8;
2
3  contract RevocationRegistry {
4
5     mapping(bytes32 => mapping(address => uint)) private revocations;
6
7     function revoke(bytes32 digest) public {
8        require (revocations[digest][msg.sender] == 0);
9        revocations[digest][msg.sender] = block.number;
10       emit Revoked(msg.sender, digest);
11    }
12
13    function revoked(address issuer, bytes32 digest) public view returns (uint) {
14       return revocations[digest][issuer];
15    }
16
17    event Revoked(address issuer, bytes32 digest);
18 }
```

The smart contract RevocationRegistry provides two functions:

- a public function to revoke a credential;
- a public function to check if a credential is in the revocation list, i.e., it has been revoked.

The revoke function takes as input a string of 32 characters, and writes a record associating it with the sender of the transaction, i.e., the address commits the line. In order to always have 32 characters, the credential before being marked on the smart contract is processed by a SHA-3 cryptographic hash algorithm and the 32-character digest is written on the smart contract.

The data structure of the smart contract is a private array of digest-address associations, named revocations (line 5). Whenever a credential is added to the register, it is mapped via the credential digest and the issuer of the transaction, i.e., the message sender. On that mapping, the block number is written, i.e., the transaction counter id.

As an implementation choice, it was decided that only the service that issued a verifiable credential can revoke it. This is to prevent third parties from revoking verifiable credentials that they have not issued. In fact, it is reasonable that only the provider who grants access to the service can eventually revoke it.

As it is possible to notice from the smart contract code, another requirement to be able to add a verifiable credential in the smart contract is the fact that it is not already present in the register (line 8), i.e., in the corresponding mapping there is not a block number indicating which transaction added the credential. If it has not already been added, then it is possible to write it (line 9). When the transaction is successfully added, an event is emitted (line 11), which communicates the issuer of the transaction and the digest of the newly added credential in the register (line 17).

A possible problem is the fact that this smart contract trusts that what is written to the registry is actually a valid digest of a credential in JWT format. In this implementation, there is no kind of access control list that allows only some addresses to write in the smart contract. In fact, once a smart contract is deployed in blockchain, its public methods can be called from any valid address. It is therefore possible that any address can call these methods and write non-consistent information to the register. This problem can be solved with a list of trusted issuers of transactions. In fact, it is possible to consider an issuer as trusted if it also implements the correct cryptographic hash algorithms on the verifiable credential before writing it to the register. These security enhancements will be added at a later stage of the i3-Market project, when this microservice will be in a production stage.

4.1.4 Verify a credential

The verification is the process of evaluation of a verifiable credential, in order to determine whether it is authentic and timely valid for the issuer or the presenter. This process includes the following checks:

- the credential conforms to the specification;
- the proof method is satisfied, i.e., the cryptographic mechanism used to prove that the information in a verifiable credential was not tampered;
- the credential is not marked as revoked in the smart contract registry.

The uport-credential library provides the methods for the first two checks, while for the third it is necessary to implement a call to the smart contract registry. The flow for verifying a credential is described in Figure 4.9.

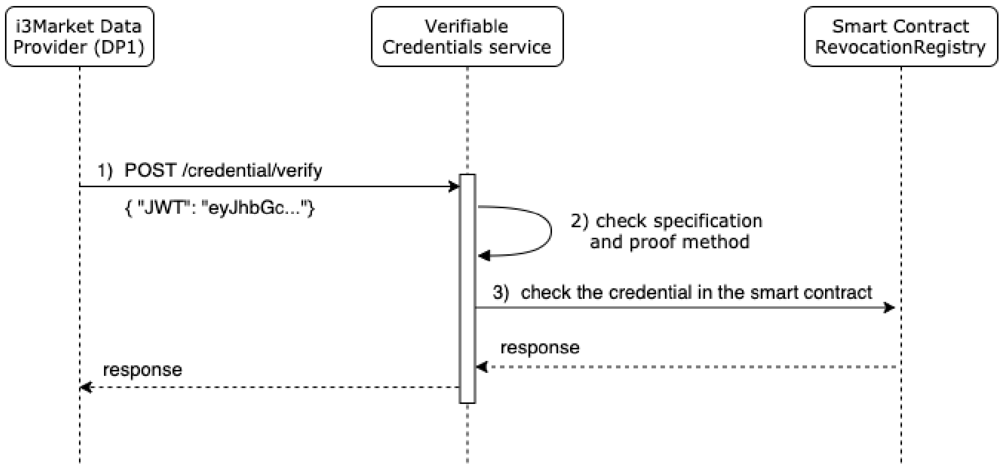

i3Market Data Provider (DP1)
Verifiable Credentials service
Smart Contract RevocationRegistry
1) POST /credential/verify
{ "JWT": "eyJhbGc..."}
2) check specification and proof method
3) check the credential in the smart contract
response
response

Figure 4.9: Verify credential workflow

In the implemented solution, in step 1) the data provider calls the Verifiable Credentials microservice, specifying the credential in JWT format to be verified in the request body. Since verifying the presence of a verifiable credential on the registry is an operation that does not change the status of the credential, this can be done by any instance of the microservice. In step 2) the Verifiable Credentials microservice checks that the issuer is valid and that the credential is in a format that complies with the data provider's specifications. If there is no problem with the credential, then it computes the hash of the credential using a SHA-3 cryptographic hash algorithms, which produces a 32-character digest. Then, in step 3), the Verifiable Credential microservice calls the 'revoked' method of the smart contract registry, specifying the issuer of the verifiable credential and the 32-character digest. The credential issuer is specified in the JWT and since only the issuer of a credential has the permissions to revoke it, it is sufficient to check that only his address, associated with the credential, is not present in the register. As it is possible to see in the solidity code of the smart contract, detailed in Section 4.1.3, this method returns the block number when it was revoked by 'issuer', or 0 if it was not. In this way it is possible to know if the credential has been revoked or not. This information is then returned as a response to the data provider, who will decide for himself what the next steps will be, for example requesting the issuance of a new valid credential or informing the user that he can no longer request access to that data or service.

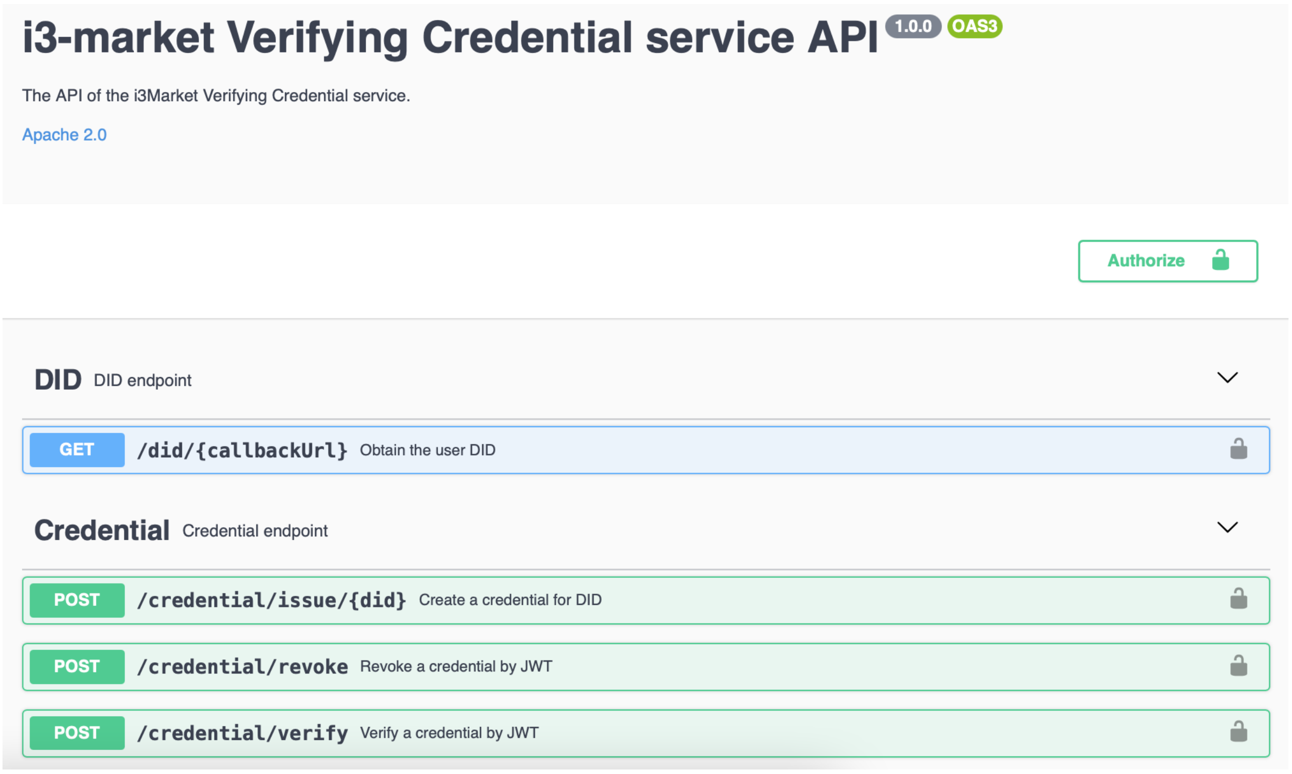
This API is used in the integration of the OIDC identity provider. In fact, to authenticate a user on the basis of the revealed credentials, a further check on the registry is necessary to ensure that the credential is not revoked. This part is described in Section 5.1.

4.2 Deployment

Docker
Docker [8] is an open platform for developing, shipping, and running applications. Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow developers to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so it is not needed to rely on what is currently installed on the host. The solution is deployed via Docker.

Ngrok
Ngrok [21] is a cross-platform reverse proxy server that allows developers to expose a local server, located behind NAT, and firewalls to the Internet via a secure ssh tunnel. When an uPort credential in QR code format is generated, it could contains a callbackUrl, which is the endpoint where the acknowledge will be sent after the credential is accepted by the user. During development, the Verifiable Credential microservice runs on localhost. So the acknowledge cannot return from the uPort app to localhost, because localhost is not exposed as public URL. For this reason the service is exposed through Ngrok on a self-generated public address. In a production environment this configuration is not necessary, since a public server is reachable by the callback acknowledge from the uPort mobile app, so this problem will not exist.

Swagger
The implemented solution is a RESTful microservice that allows the management of credentials through APIs. The APIs are documented using Swagger [25]. Swagger is the most widely used tooling ecosystem for developing APIs with the OpenAPI Specification (OAS). It consists in a set of rules, i.e., a specification for a format describing REST APIs. The format is both machine-readable and human-readable. As a result, it can be used to share documentation among product managers, testers and developers, but can also be used by various tools to automate API-related processes. The specification is described in Figure 4.10.


i3-market Verifying Credential service API 1.0.0 OAS3

The API of the i3Market Verifying Credential service.

Apache 2.0

DID DID endpoint

GET /did/{callbackUrl} Obtain the user DID

Credential Credential endpoint

POST /credential/issue/{did} Create a credential for DID

POST /credential/revoke Revoke a credential by JWT

POST /credential/verify Verify a credential by JWT

Figure 4.10: Swagger specification of Verifiable Credential microservice

In particular, the Swagger specification in Figure 4.10 describes the following APIs:

- GET /did/{callbackUrl}: This API is a utility. It generates a QR code to be scanned, after that the identity will be resolved and the user's DID returned to the callbackUrl via query parameter. This API is used in the registration process described in Section 4.1.1, to retrieve the user's DID in order to create a verifiable credential to define whether the user is a data consumer or a data provider.
- POST /credential/issue/{did}: This API is responsible for creating a verifiable credential. The credential information is passed via the body parameter. In the case of registration, the body is {*"data_consumer": true*}, in a purchase case it could be more structured, for example:

```
{
   "userDid": "ethr: did: 0xa54 .....",
   "service": "data_access",
   "dataset": "data_set1",
   "serviceExpiryDate": "16843234"
}
```

  The DID parameter is relative to the DID of the user to which the verifiable credential must be associated.
- POST /credential/verify: This API is responsible for verifying a verifiable credential. The verifiable credential is passed in string format in the body parameter.
- POST /credential/revoke: This API is responsible for revoking a verifiable credential. The verifiable credential is passed in string format in the body parameter.

5 Integration

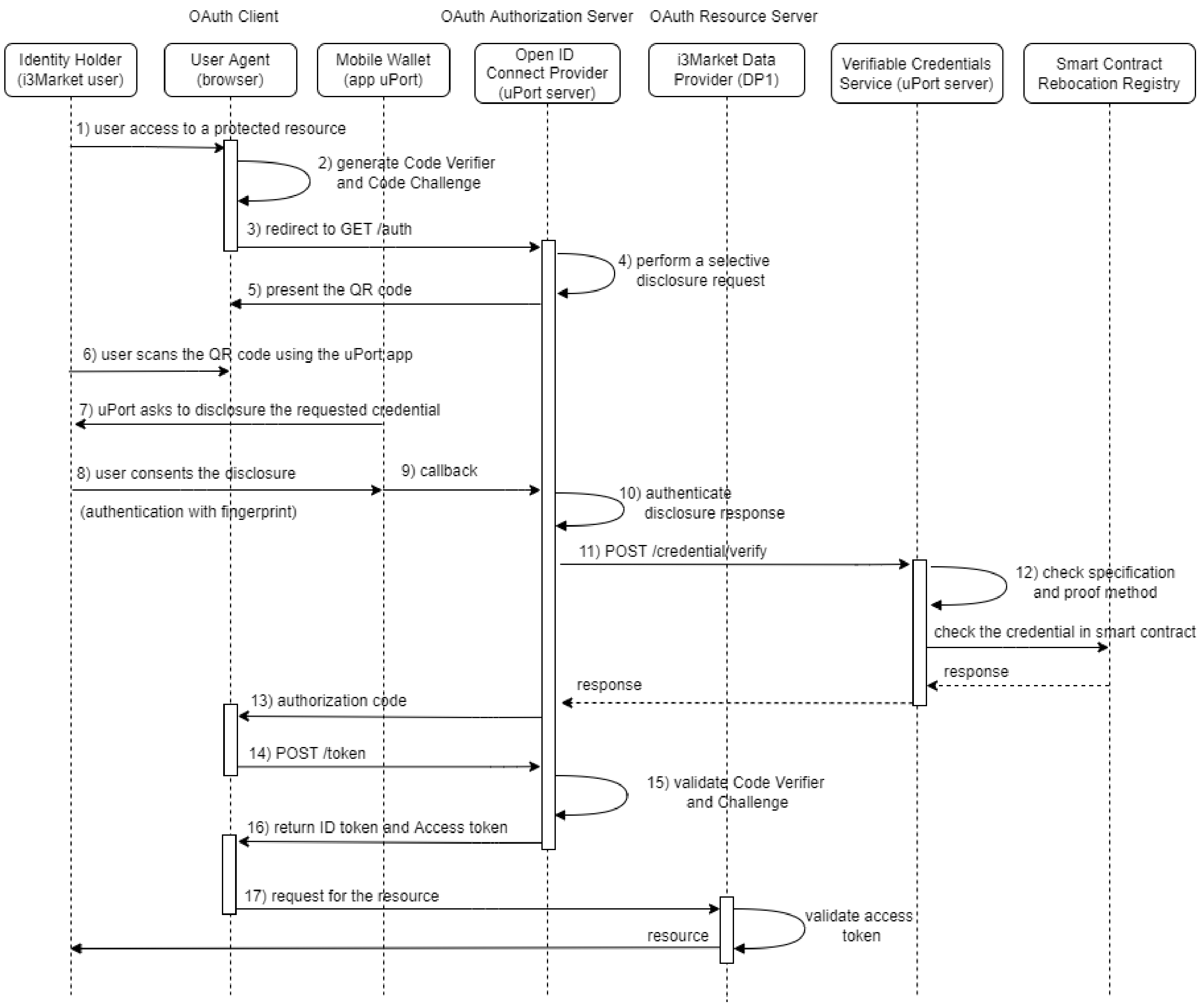
This chapter presents some possible integrations of the Verifiable Credential microservice with other components of the i3-Market architecture. The use of verifiable credentials allows the distributed and decentralized management of users. In particular, users can use verifiable credentials issued as a certificate to obtain a token necessary to access specific services or protected resources within the marketplace. In order to verify the verifiable credentials and use them in an authentication process, we integrated an open source Open ID Connect provider [23] with uPort. In this way, users can be authenticated based on the verifiable credentials they hold. Section 5.1 describes the Open ID Connect provider and the integrations that were required to implement the authorization code flow + PKCE with verifiable credentials. Section 5.2 describes the integration of the Verifiable Credential microservice with a custom software wallet that will replace the uPort mobile application in the later stages of the project.

5.1 Open ID Connect provider

As presented in Section 4.1, verifiable credentials are generated by the Verifiable Credential microservice and stored in the uPort user wallet. Verifiable credentials are also used in the login on services and to access resources available in the various data providers. When accessing i3-Market services, users are asked to disclosure the requested verifiable credentials to the OAuth Authorization Server, which plays the role of verifier in the verifiable credentials ecosystem. The verifier is therefore an OAuth 2.0 Authorization Server with Open ID Connect that implements the Authorization Code + PKCE flow. As described in Section 2.1, there are many entities involved in an OAuth 2.0 authentication flow. In relation to the use case of the project, they are the following:

- the Resource Owner is the i3-Market data consumer.
- the Client is the frontend of the data provider websites.
- the User Agent is the browser.
- the Resource Server is played by the i3-Market data providers.
- the Authorization Server is the Open ID Connect Provider, that is played by the i3-Market data provider.

The Open ID Connect Provider needs to be able to verify whether the credential is genuine and tamper free. For this reason, the Open ID Connect Provider must be able to easily validate whether a presented credential has been revoked or not throught the Verifiable Credential microservice. In order to do that, an integration with the Verifiable Credential microservice has been developed. The authentication flow used in i3-Market is shown in Figure 5.1. It is basically a traditional Authorization Code + PKCE process, but some changes have been made. In fact, the Open ID Connect provider requests verifiable credentials as scope and performs a verification of the verifiable credentials through its instance of the Verifiable Credential microservice using the API *verify*, previously described in Section 4.1.4.



OAuth Client
OAuth Authorization Server
OAuth Resource Server
Identity Holder (i3Market user)
User Agent (browser)
Mobile Wallet (app uPort)
Open ID Connect Provider (uPort server)
i3Market Data Provider (DP1)
Verifiable Credentials Service (uPort server)
Smart Contract Rebocation Registry
1) user access to a protected resource
2) generate Code Verifier and Code Challenge
3) redirect to GET /auth
4) perform a selective disclosure request
5) present the QR code
6) user scans the QR code using the uPort app
7) uPort asks to disclosure the requested credential
8) user consents the disclosure
(authentication with fingerprint)
9) callback
10) authenticate disclosure response
11) POST /credential/verify
12) check specification and proof method
check the credential in smart contract
response
response
13) authorization code
14) POST /token
15) validate Code Verifier and Challenge
16) return ID token and Access token
17) request for the resource
validate access token
resource

Figure 5.1: Authorization Code + PKCE flow with verifiable credentials

In step (1), the user wants to access a resource or service in the marketplace. The resources and services are made available by data providers, who expect to receive a valid access token and id token, with the necessary scope to access the resource or service. So, the first thing a data provider website does is to initialize the authentication flow (2). The authentication with Authorization Code flow + PKCE is done through an OAuth 2.0 SDK [15], which first generates a code verifier and a code challenge. Specifically, the OAuth 2.0 SDK creates a cryptographically-random *code_verifier* and from this generates a *code_challenge*. After that, the authorization code + PKCE flow is initialized with the first call */authorize*. The main difference of this Open ID Connect provider compared to traditional ones is that it requires the disclosure of verifiable credentials. To specify the credentials to be revealed, the scope field is used. The Open ID Connect provider has the following static scopes:

- openid: Mandatory for the Open ID Connect standard. It returns the sub field of the *id_token*, and its value is the user did.
- profile: It adds information about the user profile into the *id_token*.
- vc: It adds the field *verifiable_claims* into the *id_token*. Useful when the Open ID Connect Provider wants to check any information about the verifiable claims asked.

Compared to the standard scope of Open ID Connect, the scopes added are *vc* and *vce*. On the other hand, the standard *email* scope, which returns the user's email, is not present.

There are two different types of scopes:

- vc:vc_name: It asks the user for the specific verifiable claim *vc_name*. It is optional, so the user can decide whether to append it or not. If the issuer of the verifiable claim is not trusted, it will be added into untrusted verifiable claims array of the *id_token*. These arrays are described in the end of this section.
- vce:vc_name: It asks the user for the essential verifiable claim with name: *vc_name*. It is mandatory, so if the user does not provide it or the issuer is untrusted, the login and consent process will be cancelled.

After specifying in the scope field which credentials need to be disclosed, the OAuth SDK initializes the authentication process, performing the call to the */authorize* endpoint (3). The Open ID Connect provider performs a selective disclosure request (4), using the uPort libraries, and displays a QR code on the login page in the browser (5), as shown in Figure 5.2.

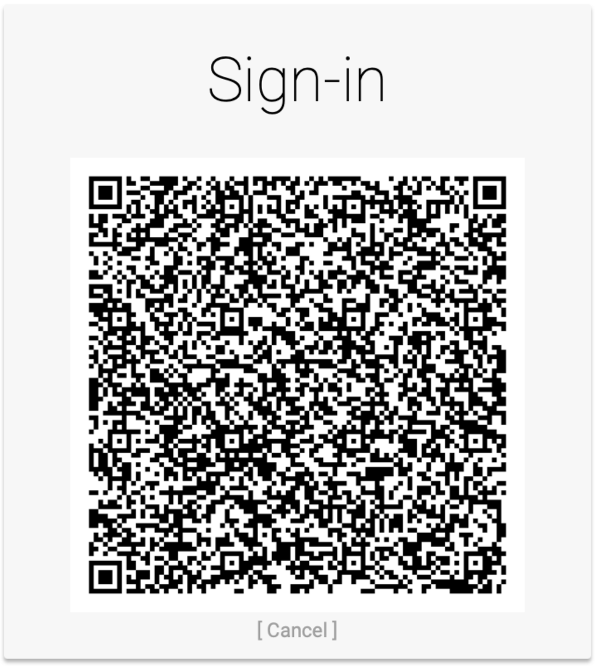
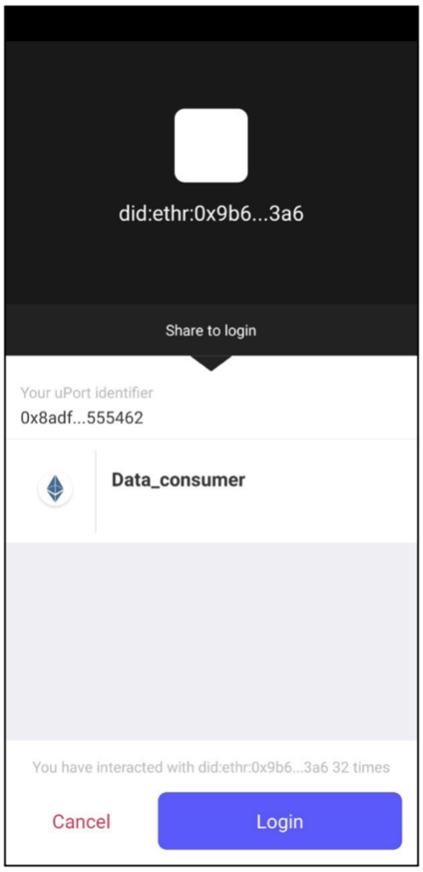


Figure 5.2: Sign In QR code

The user must then scan the QR code through their uPort app (6). At this point a notification will appear on the app with the authentication request, specifying the credentials that must be revealed (7), as shown in Figure 5.3. Pressing on Login, the fingerprint will be asked to authorize the disclosure of the credentials (8).



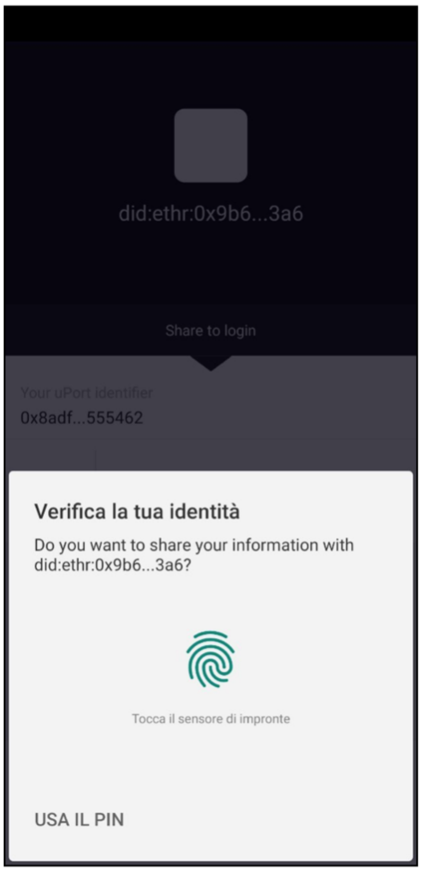


Figure 5.3: Disclosure of the *data_consumer* credentials and login with fingerprint

After the disclosure of the required credentials, a callback to the Open ID Connect provider (9) follows. The Open ID Connect provider performs an *authenticateDisclosureResponse*, i.e., it checks if all the required verifiable credentials are present and if the verifiable credential issuer is trusted (10). Subsequently it remains to check that the credentials are valid and not revoked. This is why the Verifiable Credential microservice has been integrated with the Open ID Connect provider. In particular, the credentials are sent in JWT format through the verify (11) API, which checks that they have not expired, calculates the hash and checks if they are present in the Revocation Registry (12). It then returns the array of revoked or invalid credentials as a response. If the response array is empty, then all credentials are valid. If all credentials are valid, the Open ID Connect provider returns the authorization code to the OAuth SDK of the i3-market data provider (13). The Data Provider SDK performs POST */token* (14). The *code_verifier* and *code_challenge* are checked (15) and the *id_token* and the *access_token* are returned to the Data Provider website (16). Now that the Data Provider website has a valid *access_token*, it can get the resource (17).

When the authorization and authentication process finishes, two tokens are returned: *access_token* and *id_token*. Both tokens have the following fields:

- sub: the identity (user) did
- iat: issued at
- exp: the expiration date
- iss: the token issuer (OIDC Provider)
- aud: the audience (who or what the token is intended for)

The *access_token* can be used to prove that the user consents access for a specific scope. Following example shows a decoded access token:

```
{
  "jti": "Jrg0pJEtv6gVvH6q8C_ST",
  "sub": "did:ethr:0x8adfb8543c3c14b83c3a104ed8aeb19b64555462",
  "iat": 1612171015,
  "exp": 1612174615,
  "scope": "openid vc vce:consumer",
  "iss": "https://localhost:3000",
  "aud": "oidcCLIENT"
}
```

The other specific parameters for the *access_token* are the following:

- jti: unique identifier for the JWT. It is used to prevent the JWT from being replayed.
- scope: the scope requested in the authorization request

The *id_token* contains information about the identity. The important field of this token is:

- verifiable_claims: it represent an arrays of trusted and untrusted verifiable credentials.

Following example shows a decoded *id_token*:

```
{
  "sub": "did:ethr:0x8adfb8543c3c14b83c3a104ed8aeb19b64555462",
  "verified_claims": {
    "trusted": [
      {
        "iat": 1611572251,
        "exp": 1614164251,
        "sub": "did:ethr:0x8adfb8543c3c14b83c3a104ed8aeb19b64555462",
        "claim": {
          "consumer": true
        },
        "iss": "did:ethr:0xe6f2be80ed5521529f67b39a74aa428282cf0312",
        "jwt": "eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1Nks..."
      }
    ],
    "untrusted": []
  },
  "at_hash": "yZghpdiOSEBSdbp0NPE27A",
  "aud": "oidcCLIENT",
  "exp": 1612174615,
  "iat": 1612171015,
  "iss": "https://localhost:3000"
}
```

Through the *id_token* it is possible to know which of the revealed verifiable credentials are verified (trusted) or not (untrusted).

5.2 Software wallet

uPort technology allows users to create verifiable credentials, save them in the uPort mobile wallet application and reveal them through selective disclosure requests. However, this technology is limiting, as it depends on an external service. The uPort app despite being open source is difficult to customize according to any needs or new developments. For these reasons, in the second release of the project we decided to replace the uPort wallet with a custom software wallet, which consists of a desktop application.

To implement the features that uPort guaranteed, we decided to use the Veramo framework [11], which in addition to looking like a promising technology, gave us guarantees on integration with the components already developed. In fact, it shares the DID resolver with uPort which allowed us to keep many components almost unchanged. The Veramo framework provides APIs to implement the flows of identity creation, storing of credentials in the wallet, disclousure of verifiable credentials, which allowed us to replicate the functionalities of the uPort mobile application [9].

I will not go into the functional part of the wallet as it was developed by a project partner. What I am going to describe is my contribution, which consists in integrating the wallet with the Verifiable Credentials microservice described in the previous chapter. In particular, the integration consists in some changes regarding the issuance of verifiable credentials. The other flows such as verification, revocation, and so on remained unchanged as they did not depend on the uPort mobile application. The flow of issuing verifiable credentials using the Veramo framework is shown in Figure 5.4.

The initial part of the flow is very similar to what is presented in Section 4.1.2. We consider the flow in which a verifiable credential is issued following a purchase on the website of a data provider in the marketplace. While browsing a marketplace website, the user makes a purchase. Again, the purchase is notarized through a smart contract and then the data provider triggers the issuing of a verifiable credentials that attests that particular purchase. In the uPort solution, a credential was created through a POST API call, communicating the user's DID and the claims that are represented by the verifiable credential. The DID was communicated by the i3-Market data provider website because the user was already logged into the system. In the software wallet there is much more flexibility because it is disconnected from the uPort solution and therefore it is possible to develop an ad hoc wallet, according to the specific purpose of the solution. In fact, in the wallet solution with Veramo there is not a mobile application that scans a QR code, as it was a requirement due to the uPort technology that works in that way. Therefore, the creation of the credential does not save it in a mobile application, but in a desktop application. The simplest way to communicate between a service deployed in the cloud, i.e. the Verifiable Credential microservice, and a desktop application that runs on localhost is throught the use of the browser.

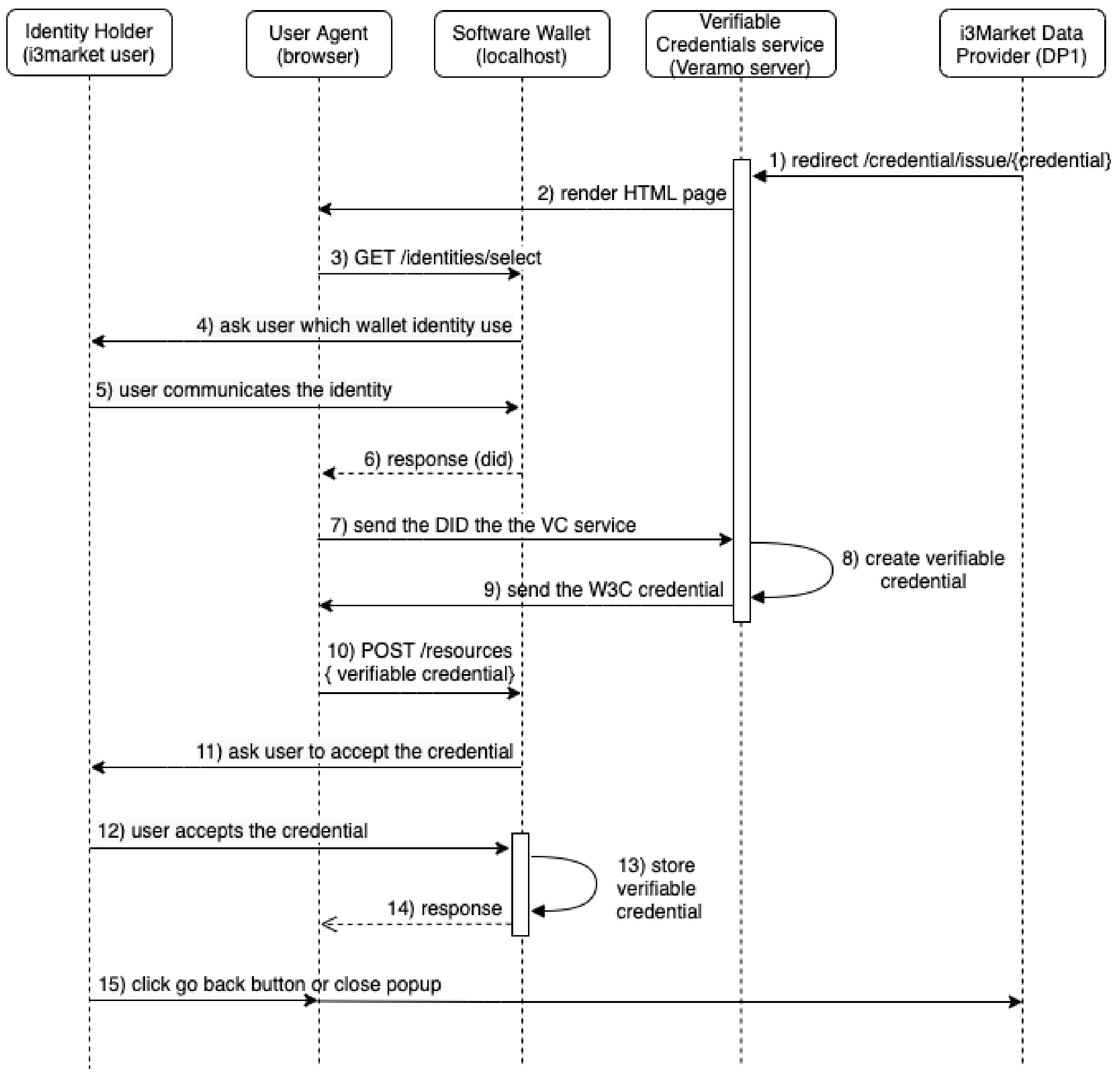

Identity Holder (i3market user)
User Agent (browser)
Software Wallet (localhost)
Verifiable Credentials service (Veramo server)
i3Market Data Provider (DP1)
1) redirect /credential/issue/{credential}
2) render HTML page
3) GET /identities/select
4) ask user which wallet identity use
5) user communicates the identity
6) response (did)
7) send the DID the the VC service
8) create verifiable credential
9) send the W3C credential
10) POST /resources { verifiable credential}
11) ask user to accept the credential
12) user accepts the credential
13) store verifiable credential
14) response
15) click go back button or close popup

Figure 5.4: Issue credential workflow with Veramo framework

As can be seen from Figure 5.4, initially a data provider requests the issuance of a verifiable credential through a redirect on a url of the Verifiable Credential microservice (1). This URL is actually a GET API that returns an HTML page as a response. The user navigating on the link will then be directed to an HTML page executed by the browser, which can call other applications that run on the localhost, such as the wallet software application. Verifiable credential claims are transmitted via URL parameter in an URL encoded format. An example of claims and their URL encoding are shown below.

```
// Claims:
{
  "role": "provider",
  "company": "GFT Italia"
}

// Encoded claims:
%7B%22role%22%3A%22provider%22%2C%22company%22%3A%22gft%22%7D
```

When the page is rendered by the browser (2), an XMLHttpRequest call is automatically initiated on the wallet to the API *GET /identities/select* (3). This API triggers a notification on the wallet, requesting to share the user's identity (4). The user then approves the sharing of his identity (5), i.e.

the DID, which is returned as a response to the HTML page (6). As soon as a response is received, the page calls submits the user's DID to the Verifiable Credential microservice (7) which through the Veramo framework generates the verifiable credential and signs it with its own private key (8). The signed verifiable credential is returned to the HTML page (9), which as soon as it receives it forwards it to the wallet via the *POST /resources* API (10). This API triggers a second notification to the wallet, which asks the user to accept and save the verifiable credential (11). The user approves saving of the verifiable credential (12). The wallet then saves the verifiable credential (13) in its database and responds with the outcome of the operation to the HTML page, which shows it to the user, providing feedback on the operation. The user can click a go-back button to return to the website of the data provider in the marketplace (15).

6 Conclusions and future work

This thesis describes the implementation of a service to issue, verify and revoke verifiable credentials in a self-sovereign identity architecture. This service is already used within the i3-Market project and it is already integrated with some other components such as the OpenID Connect Provider authorization server and the i3-Market software wallet. As i3-Market is a three-year H2020 project, this component will inevitably evolve further in the coming months, based on the new features that will be needed to better manage the digital identity and integration with the other components of the architecture that are under development. In this sense, the implemented microservice has already evolved a lot since I started writing the thesis. I made the first implementation using the uPort technology to issue and verify verifiable credentials and the uPort mobile wallet to store them, as we did not yet have our custom wallet available. Subsequently, when the i3-Market wallet reached a minimum viable product phase, I was able to integrate it into the flow of issuing the verifiable credentials, thus allowing to save the credentials in a wallet that uses Veramo technology. It is not excluded that in the future a better technology will come out and therefore it can be further extended with another framework. As regards the management of verifiable credentials, there are still some aspects to improve as future work. In particular, in use cases such as i3-Market, there should be a procedure to recover the verifiable credentials of a user in the event that the wallet or private key is lost. To do this, there should be a cloud wallet that supports the backup of user credentials, so that they can be recovered. Another aspect for the management of verifiable credentials consists in the possibility of updating one's own DID, for example by changing the ownership of a DID or by delegating the possibility to a user of act on behalf of another identity, thus using verifiable credentials not associated with him. Many of these aspects will be covered in future releases of the project and therefore they have not discussed in the thesis.

Both uPort and Veramo have proven to be valid technologies for implementing ecosystems based on self-sovereign identity. uPort has a mobile wallet that allows users to store verifiable credentials and reveals them during the authentication and authorization processes very quickly and immediately, by scanning a QR code. The wallet app is open source and can be downloaded from the play store and app store for free. Veramo is a more complex framework, but also much more flexible. It consists of a framework to support the creation of agents, each of which may have different capabilities. It does not have a ready-made wallet, so you need to implement it yourself. Using Veramo, to design a self-sovereign identity ecosystem it is therefore necessary to develop more agents. In particular, an agent was needed for the verifiable credential service, with the ability to issue, verify and revoke verifiable credentials. Then an agent for the custom wallet, with the capabilities to create identities, store verifiable credentials and disclose through verifiable presentation. Finally an agent for the authorization server, with the capability of generating disclosure requests to be sent to the wallet. Although uPort is more straightforward, Veramo is more flexible, allows you to develop custom solutions and above all independent from external services, such as the uPort wallet and the servers behind it.

By project requirement, the reference blockchain is Ethereum, specifically using the Hyperledger Besu open source client. For this reason we had to use a technology that was compatible with this blockchain. However, there are many other open source frameworks to implement services and agents to support self-sovereign identity ecosystems. Hyperledger Aries is certainly one of the most discussed and popular. Although it was initially considered and analyzed, it was not possible to use it as it is 100% compatible only with Hyperledger Indy and does not implement the resolver for our project's blockchain client. Despite this, in future projects it is certainly an interesting solution that is worth analyzing.

The whole thesis focuses on the authentication and access management of two types of users within a marketplace. However, self-sovreign identity can be much more. In fact, people may not be the only ones who can benefit from this model for identity. A much discussed and current topic concerns the IoT [5]. A lot of technology is now connected on the Internet and there are many use cases where providing them with an identity would be useful. It would therefore be interesting to explore how to use this identity model to allow IoT devices to authenticate and be authorized to perform actions regarding their credentials.

Appendix A

Example of DID document with 1 verification method type

```
{
    "@context": [
      "https://www.w3.org/ns/did/v1",
      "https://w3id.org/security/suites/ed25519-2020/v1"
    ],
    "id": "did:example:123",
    "authentication": [
      {
        "id": "did:example:123#z6MkecaLyHuYWkayBDLw5ihndj3T1m6zKTGqau3A51G7RBf3",
        "type": "Ed25519VerificationKey2020", // external (property value)
        "controller": "did:example:123",
        "publicKeyMultibase": "zAKJP3f7BD6W4iWEQ9jwndVTCBq8ua2Utt8EEjJ6Vxsf"
      }
    ],
    "capabilityInvocation": [
      {
        "id": "did:example:123#z6MkhdmzFu659ZJ4XKj31vtEDmjvsi5yDZG5L7Caz63oP39k",
        "type": "Ed25519VerificationKey2020", // external (property value)
        "controller": "did:example:123",
        "publicKeyMultibase": "z4BWwfeqdp1obQptLLMvPNgBw48p7og1ie6Hf9p5nTpNN"
      }
    ],
    "capabilityDelegation": [
      {
        "id": "did:example:123#z6Mkw94ByR26zMSkNdCUi6FNRsWnc2DFEeDXyBGJ5KTzSWyi",
        "type": "Ed25519VerificationKey2020", // external (property value)
        "controller": "did:example:123",
        "publicKeyMultibase": "zHgo9PAmfeoxHG8Mn2XHXamxnnSwPpkyBHAMNF3VyXJCL"
      }
    ],
    "assertionMethod": [
      {
        "id": "did:example:123#z6MkiukuAuQAE8ozxvmahnQGzApvtW7KT5XXKfojjwbdEomY",
        "type": "Ed25519VerificationKey2020", // external (property value)
        "controller": "did:example:123",
        "publicKeyMultibase": "z5TVraf9itbKXrRvt2DSS95Gw4vqU3CHAdetoufdcKazA"
      }
    ]
}
```

Example of DID Document with many different key types

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/security/suites/jws-2020/v1"
  ],
  "verificationMethod": [
    {
      "id": "did:example:123#key-0",
      "type": "JsonWebKey2020",
      "controller": "did:example:123",
      "publicKeyJwk": {
        "kty": "OKP", // external (property name)
        "crv": "Ed25519", // external (property name)
        "x": "VCpo2LMLhn6iWku8MKvSLg2ZAoC-nlOyPVQaO3FxVeQ" // external (property name)
      }
    },
    {
      "id": "did:example:123#key-1",
      "type": "JsonWebKey2020",
      "controller": "did:example:123",
      "publicKeyJwk": {
        "kty": "OKP", // external (property name)
        "crv": "X25519", // external (property name)
        "x": "pE_mG098rdQjY3MKK2D5SUQ6ZOEW3a6Z6T7Z4SgnzCE" // external (property name)
      }
    },
    {
      "id": "did:example:123#key-2",
      "type": "JsonWebKey2020",
      "controller": "did:example:123",
      "publicKeyJwk": {
        "kty": "EC", // external (property name)
        "crv": "secp256k1", // external (property name)
        "x": "Z4Y3NNOxv0J6tCgqOBFnHnaZhJF6LdulT7z8A-2D5_8", // external (property name)
        "y": "i5a2NtJoUKXkLm6q8nOEu9WOkso1Ag6FTUT6k_LMnGk" // external (property name)
      }
    },
    {
      "id": "did:example:123#key-3",
      "type": "JsonWebKey2020",
      "controller": "did:example:123",
      "publicKeyJwk": {
        "kty": "EC", // external (property name)
        "crv": "secp256k1", // external (property name)
        "x": "U1V4TVZVMUpUa0ZVU1NBcU9CRm5IbmFaaEpGNkxkdWx", // external (property name)
        "y": "i5a2NtJoUKXkLm6q8nOEu9WOkso1Ag6FTUT6k_LMnGk" // external (property name)
      }
    },
    {
      "id": "did:example:123#key-4",
      "type": "JsonWebKey2020",
      "controller": "did:example:123",
      "publicKeyJwk": {
        "kty": "EC", // external (property name)
        "crv": "P-256", // external (property name)
        "x": "Ums5WVgwRkRTVVFnU3k5c2xvZllMbEcwM3NPRW91ZzN", // external (property name)
        "y": "nDQW6XZ7b_u2Sy9slofYLlG03sOEoug3I0aAPQ0exs4" // external (property name)
      }
    },
    {
      "id": "did:example:123#key-5",
      "type": "JsonWebKey2020",
      "controller": "did:example:123",
      "publicKeyJwk": {
        "kty": "EC", // external (property name)
        "crv": "P-384", // external (property name)
        "x": "VUZKSlUwMGdpSXplekRwODhzX2N4U1BYdHVYWUZsaXVDR25kZ1U0UXA4", // external (property name)
        "y": "jq4QoAHKiIzezDp88s_cxSPXtuXYFliuCGndgU4Qp8l91xzD1spCmFIzQgVjqvcP" // external (property name)
      }
    },
    {
      "id": "did:example:123#key-6",
      "type": "JsonWebKey2020",
      "controller": "did:example:123",
      "publicKeyJwk": {
        "kty": "EC", // external (property name)
        "crv": "P-521", // external (property name)
        "x": "VTI5c1lYSmZWMmx1WkhNZ0dQTXhaYkhtSnBEU3UtSXZwdUtpZ0VOMnB6Z1d", // external (property name)
        "y": "UW5WNVgwSnBkR052YVc0Z1VqY1B6LVpoZWNaRnliT3FMSUpqVk9sTEVUSDd" // external (property name)
      }
    },
    {
      "id": "did:example:123#key-7",
      "type": "JsonWebKey2020",
      "controller": "did:example:123",
      "publicKeyJwk": {
        "kty": "RSA", // external (property name)
        "e": "AQAB", // external (property name)
        "n": "UkhWaGJGOUZRMTlFVWtKSElBdENGV2hlU1F2djFNRXh1NVJMQ01UNGpWazlr
        aEpLdjhKZU1YV2UzYldIYXRqUHNrZGYyZGxhR2tXNVFqdE9uVUtMNzQybXZyNHRDb
        GRLUzNVTElhVDFoSkluTUhIeGoyZ2N1Yk82ZUVlZ0FDUTRRU3U5TE8wSC1MTV9MM0R
        zUkFCQjdRamE4SGVjcHl1c3BXMVR1X0RicXhjU253ZW5kYW13TDUyVjE3ZUtobE80dVh3dj
        JIRmx4dWZGSE0wS21DSnVqSUt5QXhqRF9tM3FfX0lpSFVWSEQxdERJRXZMUGhHOUF6c24zajk1ZC1zYU" // external (property name)
      }
    }
  ]
}
```

Appendix B

Example of a simple Verifiable Credential

```
{
  // set the context, which establishes the special terms we will be using
  // such as 'issuer' and 'alumniOf'.
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "http://example.edu/credentials/1872", // specify the identifier for the credential
  "type": ["VerifiableCredential", "AlumniCredential"], // the credential types, which
      declare what data to expect in the credential
  "issuer": "https://example.edu/issuers/565049", // the entity that issued the credential
  "issuanceDate": "2010-01-01T19:73:24Z", // when the credential was issued
  "credentialSubject": { // claims about the subjects of the credential
    "id": "did:example:ebfeb1f712ebc6f1c276e12ec21", // identifier for the only subject of
        the credential
    "alumniOf": {   // assertion about the only subject of the credential
      "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
      "name": [{
        "value": "Example University",
        "lang": "en"
      }, {
        "value": "Exemple d'Universite",
        "lang": "fr"
      }]
    }
  },
  // digital proof that makes the credential tamper-evident
  "proof": {
    "type": "RsaSignature2018",  // the cryptographic signature suite that was used to
        generate the signature
    "created": "2017-06-18T21:19:10Z", // the date the signature was created
    "proofPurpose": "assertionMethod", // purpose of this proof
    "verificationMethod": "https://example.edu/issuers/keys/1", // the identifier of the
        public key that can verify the signature
    // the digital signature value
    "jws": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..TCYt5X
      sITJX1CxPCT8yAV-TVkIEq_PbChOMqsLfRoPsnsgw5WEuts01mq-pQy7UJiN5mgRxD-WUc
      X16dUEMGlv50aqzpqh4Qktb3rk-BuQy72IFLOqV0G_zS245-kronKb78cPN25DGlcTwLtj
      PAYuNzVBAh4vGHSrQyHUdBBPM"
  }
}
```

Example of a Verifiable Presentation

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "type": "VerifiablePresentation",
  // the verifiable credential issued in the previous example
  "verifiableCredential": [{
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "id": "http://example.edu/credentials/1872",
    "type": ["VerifiableCredential", "AlumniCredential"],
    "issuer": "https://example.edu/issuers/565049",
    "issuanceDate": "2010-01-01T19:73:24Z",
    "credentialSubject": {
      "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
      "alumniOf": {
        "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
        "name": [{
          "value": "Example University",
          "lang": "en"
        }, {
          "value": "Exemple d'Universite",
          "lang": "fr"
        }]
      }
    },
    "proof": {
      "type": "RsaSignature2018",
      "created": "2017-06-18T21:19:10Z",
      "proofPurpose": "assertionMethod",
      "verificationMethod": "https://example.edu/issuers/keys/1",
      "jws": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..TCYt5X
        sITJX1CxPCT8yAV-TVkIEq_PbChOMqsLfRoPsnsgw5WEuts01mq-pQy7UJiN5mgRxD-WUc
        X16dUEMGlv50aqzpqh4Qktb3rk-BuQy72IFLOqV0G_zS245-kronKb78cPN25DGlcTwLtj
        PAYuNzVBAh4vGHSrQyHUdBBPM"
    }
  }],
  // digital signature by Pat on the presentation
  // protects against replay attacks
  "proof": {
    "type": "RsaSignature2018",
    "created": "2018-09-14T21:19:10Z",
    "proofPurpose": "authentication",
    "verificationMethod": "did:example:ebfeb1f712ebc6f1c276e12ec21#keys-1",
    // 'challenge' and 'domain' protect against replay attacks
    "challenge": "1f44d55f-f161-4938-a659-f8026467f126",
    "domain": "4jt78h47fh47",
    "jws": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..kTCYt5
      XsITJX1CxPCT8yAV-TVIw5WEuts01mq-pQy7UJiN5mgREEMGlv50aqzpqh4Qq_PbChOMqs
      LfRoPsnsgxD-WUcX16dUOqV0G_zS245-kronKb78cPktb3rk-BuQy72IFLN25DYuNzVBAh
      4vGHSrQyHUGlcTwLtjPAnKb78"
  }
}
```

Bibliography

[1] Christopher Allen. The path to self-sovereign identity. http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html, 2016.

[2] ERC-725 Alliance. Erc-725 ethereum identity standard. https://erc725alliance.org/.

[3] Auth0. Jwt. https://jwt.io/, 2020.

[4] Auth0. Oauth. https://auth0.com/docs/protocols/protocol-oauth2, 2020.

[5] Judy Bowen, Annika Hinze, and Rangi Matamua. Participatory data design: managing data sovereignty in iot solutions. Technical report, Hamilton, New Zealand, 2021. Working Paper.

[6] P. Braendgaard. What is an uPort identity? [Online]. available at: https://medium.com/uport/what-is-a-uport-identity-b790b065809c. 2017.

[7] P. Braendgaard and J. Torstensson. Eip-1056: Ethereum lightweight identity. https://eips.ethereum.org/EIPS/eip-1056, May 2018.

[8] Inc Docker. Docker. https://www.docker.com/, 2021.

[9] Veramo documentation. https://veramo.io/docs/api/core, 2021.

[10] Hyperledger Foundation. https://besu.hyperledger.org/en/stable/, 2020.

[11] Veramo framework. https://veramo.io/, 2021.

[12] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012.

[13] i3Market project. https://www.i3-market.eu, 2020.

[14] Decentralized Identity. did-jwt-vc library. https://github.com/decentralized-identity/did-jwt-vc, 2021.

[15] IdentityModel. oidc-client repository. https://github.com/IdentityModel/oidc-client-js, 2020.

[16] Michael Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.

[17] Protocol Labs. Interplanetary file system (ipfs). https://ipfs.io/, 2021.

[18] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. OAuth 2.0 Security Best Current Practice. Internet-Draft draft-ietf-oauth-security-topics-18, Internet Engineering Task Force, April 2021. Work in Progress.

[19] C. Lundkvist, R. Heck, J. Torstensson, Z. Mitton, and M. Sena. uport: A platform for self-sovreign identity. https://blockchainlab.com/pdf/uPort_whitepaper_DRAFT20161020.pdf, 2018.

[20] N. Naik and P. Jenkins. uport open-source identity management system: An assessment of self-sovereign identity and user-centric data platform built on blockchain. Available at: `https://eips.ethereum.org/EIPS/eip-1056`.

[21] Inc Ngrock. Ngrock. `https://ngrok.com/`, 2021.

[22] Wikipedia's page. List of data breaches. `https://en.wikipedia.org/wiki/List_of_data_breaches`, 2021.

[23] Panva. node-oidc-provider repository. `https://github.com/panva/node-oidc-provider`, 2020.

[24] Nat Sakimura, John Bradley, and Naveen Agarwal. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636, September 2015.

[25] SMARTBEAR. Swagger.io. `https://swagger.io/docs/`, 2020.

[26] troyhunt.com project. Have i been pwned? `https://haveibeenpwned.com/`, 2021.

[27] uPort. ethr-did repository. `https://github.com/uport-project/ethr-did`, 2020.

[28] uPort. Identity manager contract. `https://github.com/uport-project/uport-identity/blob/develop/docs/reference/identityManager.md`, 2020.

[29] W3C. Verifiable credential data model. `https://www.w3.org/TR/vc-data-model/`, 2020.

[30] W3C. Decentralized identifiers (dids) v1.0. `https://www.w3.org/TR/2021/CRD-did-core-20210627/`, 2021.