UNIVERSITÀ
DI TORINO

Dipartimento di informatica
Corso di Laurea in Informatica
Anno Accademico 2021/2022

Integrazione tra servizi di autenticazione e blockchain: come gestire wallet crittografici

Relatore:
Prof. Claudio Schifanella

Candidato:
Raoul Rondinella

Sommario

Abstract *4*

1 Introduzione *5*

1.1 Requisiti di progetto **5**

1.2 Alten **5**

1.3 Andromeda **6**

2 La Blockchain *7*

2.1 Cos'è la blockchain **7**

2.2 Algoritmi di consenso **7**

2.2.1 Proof of Work (PoW) 7

2.2.2 Proof of Stake (PoS) 8

2.2.3 Proof of Authority (PoA) 8

2.3 Tipi di wallet **8**

2.3.1 Wallet non deterministico 9

2.3.2 Wallet deterministico 10

2.3.3 Wallet gerarchico (HD Wallet) 10

2.3.4 Generazione del seme 11

2.4 Ethereum **11**

2.4.1 Smart Contracts 12

2.4.2 Gas 12

3 OAuth2 e OIDC *13*

3.1 Componenti **13**

3.1.1 Flusso di autorizzazione e attori 13

3.1.2 Scopes 14

3.1.3 Tokens 14

3.2 Open ID Connect (OIDC) **16**

3.2.1 Claims 16

3.2.2 Sicurezza 16

4 Tecnologie usate 18

4.1 Front-End 18

4.1.1 Node.js 18

4.1.2 Vue.js 18

4.1.3 Web3.js 19

4.1.4 Typescript 20

4.2 Back-End 20

4.2.1 Jakarta Persistence 20

4.2.2 Java Spring 21

5 Progettazione 22

5.1 Interazione con la dApp 22

5.2 Salvataggio dell'utente 23

5.3 Salvataggio del wallet 23

5.4 Gestione della password 24

6 Implementazione 26

6.1 Server-side 26

6.1.1 Configurazione Authorization Server 26

6.1.2 Database 27

6.2 Client-side 29

6.2.1 Quadro generale 29

6.2.2 Signup 29

6.2.3 Login 29

6.2.4 Scopes 30

6.2.5 Il mio wallet 30

6.2.6 Impostazioni account 32

Conclusioni 33

Ringraziamenti 34

Bibliografia 35

DICHIARAZIONE DI ORIGINALITÀ

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Abstract

In questo documento, esploriamo l'integrazione innovativa tra la tecnologia Blockchain e il protocollo di autorizzazione OAuth 2.0 nell'ambito di un progetto di stage presso Alten Italia.

L'obiettivo principale è quello di semplificare l'accesso e la gestione dei wallet blockchain per gli utenti non esperti, utilizzando le loro credenziali di accesso tradizionali come e-mail e password.

Ethereum, una piattaforma blockchain che supporta la creazione di smart contract, è stata selezionata come base per l'implementazione. Il progetto prevede l'integrazione di OAuth 2.0 con Ethereum, fornendo un'autenticazione sicura e semplificata, oltre alla creazione automatica del wallet per gli utenti.

Nel corso del documento, verranno discusse le tecnologie utilizzate, la progettazione e l'implementazione della soluzione proposta, mettendo in luce il valore aggiunto derivante dall'unione di queste due tecnologie per un'esperienza utente migliorata nel mondo della blockchain.

1 Introduzione

1.1 Requisiti di progetto

La crescente importanza della tecnologia blockchain e delle criptovalute ha portato all'attenzione della comunità scientifica e industriale la necessità di rendere questi strumenti accessibili a un pubblico sempre più vasto.

Tuttavia, le complesse architetture e interfacce delle piattaforme blockchain, ed una scarsa conoscenza generale di questa tecnologia, rappresentano ostacoli significativi per gli utenti comuni.

Il progetto, quindi, prova a semplificare la vita dell'utente, rendendo l'accesso alla blockchain disponibile ad un non esperto, attraverso l'integrazione di tecnologie consolidate come OAuth2 e Open ID Connect (OIDC) per la gestione dell'accesso.

Questo lavoro si propone di esaminare e sviluppare una soluzione che permetta un accesso sicuro e *user-friendly* alla blockchain, senza richiedere conoscenze tecniche avanzate da parte degli utenti.

L'idea è dare la possibilità all'utente di accedere alla rete tramite *e-mail* e *password*, senza usare le chiavi private in modo diretto.

Sarà, quindi, ancora possibile utilizzare effettivamente le chiavi private, fondamentali per una corretta interazione con la rete, ma l'utente non dovrà risentirne.

1.2 Alten

Il Gruppo ALTEN, un'azienda leader europea nel settore della consulenza per le tecnologie avanzate in ambito ingegneristico e ICT, è quotata sulla Borsa di Parigi e conta oltre 40.000 collaboratori distribuiti in 30 nazioni.

In Italia, ALTEN opera a livello nazionale con più di 2.934 collaboratori e sedi in varie città, tra cui Milano, Gallarate, Bari, Bologna, Brescia, Firenze, Genova, Napoli, Padova, Roma e Torino. L'azienda offre servizi legati all'ingegneria, all'IT, alle telecomunicazioni e al settore life sciences, vantando numerosi centri di eccellenza, tra cui digital, business intelligence, testing, formazione (ALTEN Academy), ITSM, IoT, hardware, software embedded, quality e CSV. La forza di ALTEN si basa sulla competenza e professionalità dei suoi profili tecnici e manageriali, che vengono scelti e formati attraverso percorsi strutturati e ripetibili, anche all'interno dell'ALTEN Academy, la quale offre corsi di formazione e certificazione anche all'esterno dell'organizzazione. [1]

1.3 Andromeda

Dopo tre anni di ricerca e sviluppo nel campo della Blockchain, è stata creata la piattaforma Andromeda. Il progetto mirava a sviluppare una piattaforma riutilizzabile per la gestione dei progetti aziendali legati alla Blockchain, concentrandosi in particolare sulla tracciabilità della filiera.

Andromeda affronta e risolve i tre maggiori problemi legati all'utilizzo della tecnologia Blockchain, che sono:

- L'approccio: è necessario applicare correttamente i concetti della Blockchain per sfruttarne al meglio i benefici e le potenzialità;
- La gestione: gli strumenti per il controllo e la supervisione non sono incorporati nel software Geth;
- L'infrastruttura: l'implementazione dei nodi in una rete decentralizzata e distribuita risulta complessa e onerosa.

La piattaforma Andromeda si basa sull'infrastruttura della Blockchain Ethereum e offre una soluzione compatibile con le esigenze delle imprese, facilitando l'installazione, l'amministrazione e la supervisione del sistema.

Andromeda Development Kit - ADK

Rappresenta una funzionalità che permette di applicare le metodologie di ingegneria del software aziendale tradizionale (riuso, ottimizzazione dei costi, alto disaccoppiamento) agli Smart Contract. Questa funzionalità è costituita da un insieme di strumenti, spiegati di seguito.

• Platform Contract: Smart Contract che offrono le funzionalità base standardizzate, ossia storage, access control e contract registry;

• Service Contract: Smart Contract di utilità che offrono funzionalità avanzate, ossia checkpointing e DTT;

• Librerie Java e Javascript: per l'invocazione degli Smart Contract dalle componenti off-chain. [2]

2 La Blockchain

In generale, la blockchain è un registro di transazioni che ha la particolarità di essere distribuito. Il funzionamento è quanto segue: viene creata una transazione e viene inserita in un blocco insieme ad altre transazioni, quando il blocco è relativamente della giusta dimensione viene collegato al blocco precedente.

Questo crea una vera e propria catena di blocchi dalla quale la blockchain prende il nome [3]

2.1 Cos'è la blockchain

Come anticipato, la tecnologia blockchain consiste nella creazione di un registro pubblico, distribuito ed immutabile di transazioni.

In passato la memorizzazione delle informazioni è stata affidata ad un unico *ledger* centrale che garantiva l'integrità dei dati e nel quale si riponeva una cieca fiducia.

La blockchain risolve questo problema distribuendo l'intera conoscenza delle transazioni su ogni nodo, sottoforma di "catena di blocchi".

Ogni volta che viene generata una transazione, questi viene inserita in un blocco da validare. Quando abbiamo abbastanza transazioni in un blocco un nodo si preoccuperà di validare le informazioni e, nel caso positivo, aggancerà il blocco alla "catena di blocchi". [3]

La scelta di chi valida e come validare un blocco cambia a seconda dell'implementazione.

2.2 Algoritmi di consenso

Abbiamo svariate tecniche per decidere chi ha il diritto di gestire le transazioni, ma quelle principali e/o utili al contesto sono tre: Proof of Work, Proof of Stake e Proof of Authority. [4]

2.2.1 Proof of Work (PoW)

Il Proof of Work l'algoritmo più famoso. È considerato sicuro ed affidabile ma è difficile da scalare in quanto richiede grandi quantità di energia. I *miners* competono per risolvere dei problemi computazionali, come un problema di *hashing* o di numeri primi. Il primo a trovare la soluzione al problema avrà il diritto di validare il blocco. Inoltre, riceverà una compensa in cambio, sottoforma di criptovaluta.

Nel momento in cui un *miner* trova la soluzione, la comunica agli altri nodi. Gli altri nodi potranno quindi confermare la soluzione e, nel caso in cui la soluzione si corretta, il blocco e tutte le sue transazioni saranno aggiunte nella blockchain.

Nel caso in cui uno o più *miners* trovino la soluzione contemporaneamente, creando così più rami all'interno della blockchain, verrà considerato valido il ramo con più blocchi. [5]

2.2.2 Proof of Stake (PoS)

Il Proof of Stake sta prendendo piede come alternativa al PoW. In questa alternativa i *miners* devono mettere in *staking* della criptovaluta.

Anziché lavorare per trovare una soluzione ad un problema, ora il *miner* che convaliderà la transazione viene scelto in modo casuale tra tutti i *miner*.

La probabilità, però, non è uguale per tutti i *miners*: più criptovaluta sarà messa in *staking* e per maggior tempo, maggiori saranno le probabilità di essere selezionati come validatori del blocco.

Nel caso della PoS i miners non riceveranno una ricompensa per blocco, ma una ricompensa per transazione. [6]

2.2.3 Proof of Authority (PoA)

Il Proof of Authority è un meccanismo che punta a rendere più efficiente la blockchain ma è utilizzato nelle reti blockchain private.

In questo caso i validatori non danno una garanzia ma vengono ritenuti affidabili dall'organizzazione che gestisce la blockchain.

Questo sistema sacrifica la decentralizzazione in senso classico visto che ogni nodo è in realtà gestito dalla stessa organizzazione, ma avendo un numero limitato di validatori rende il sistema più veloce. [7]

2.3 Tipi di wallet

Una premessa: nell'uso comune con il termine *wallet* ci si può riferire sia alla singola coppia "chiave privata-pubblica", sia all'insieme di coppie "chiave privata-pubblica". In questa tesi però, dato che si utilizza di frequente entrambi e rischierebbe di causare confusione, si è deciso di utilizzare *account* per riferirsi alla singola coppia e *wallet* quando ci riferiamo all'insieme di coppie (insieme di *accounts*). Questo anche per avvicinarci alle librerie che seguono, anche loro, questa nomenclatura.

Per aggiungere una nuova transazione sulla Blockchain, è necessario utilizzare un *account*. In questo modo è possibile autenticare e autorizzare le transazioni mediante un sistema di crittografia a chiave pubblica-privata.

Ogni *account* è caratterizzato da una coppia di chiavi: una chiave pubblica e una chiave privata. La chiave pubblica funge da indirizzo dell'*account* sulla blockchain e può essere

condivisa liberamente, mentre la chiave privata, responsabile della firma digitale delle transazioni, deve essere mantenuta segreta.

Quando un utente crea una transazione, viene firmata digitalmente con la propria chiave privata. La firma può essere verificata da chiunque sulla rete utilizzando la chiave pubblica associata all'account, garantendo la sua autenticità e integrità.

Più *account* possono essere raggruppati in un unico insieme chiamato *wallet*, che facilita la gestione e l'organizzazione delle risorse digitali degli utenti. [8]

2.3.1 Wallet non deterministico

È la tipologia base di *wallet*. Il *wallet* non deterministico è caratterizzato da un *set* di chiavi private generate in modo casuale.

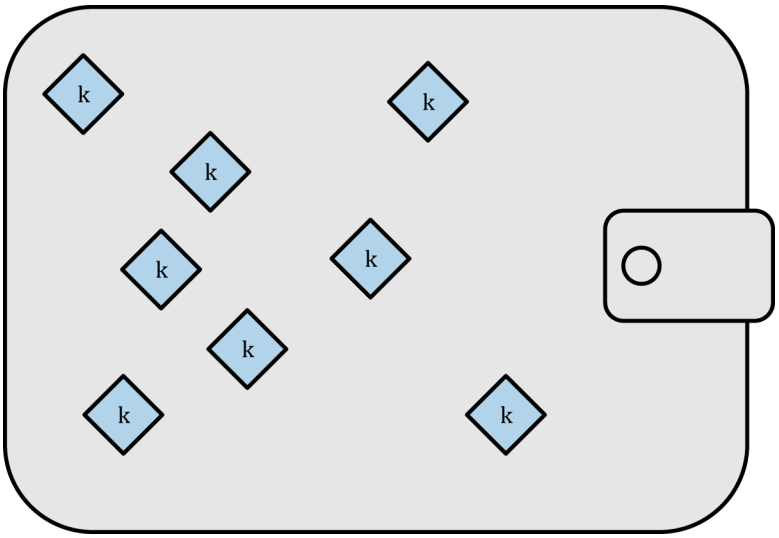


Figura 1 Wallet non deterministico. Ogni k è un account. [8]

La gestione di un wallet non deterministico può essere complicata e comporta il rischio di perdere una o più chiavi private. Prendiamo in considerazione un esempio pratico in cui un utente deve creare un wallet non deterministico.

Per iniziare, l'utente deve generare una chiave privata per ciascun account desiderato. Dopo aver creato le chiavi private, è necessario importarle nel wallet, assicurandosi di conservarle in modo sicuro e protetto. Ogni volta che l'utente aggiunge un nuovo account, deve ripetere questo processo, aumentando la complessità della gestione delle chiavi private.

Per questo motivo ne viene generalmente scoraggiato l'utilizzo.

2.3.2 Wallet deterministico

In questa tipologia del wallet, le chiavi private generate non sono indipendenti: viene generato casualmente un *seed* che verrà usato come partenza nella generazione di tutti gli *accounts* del *wallet*, con un sistema "a cascata"

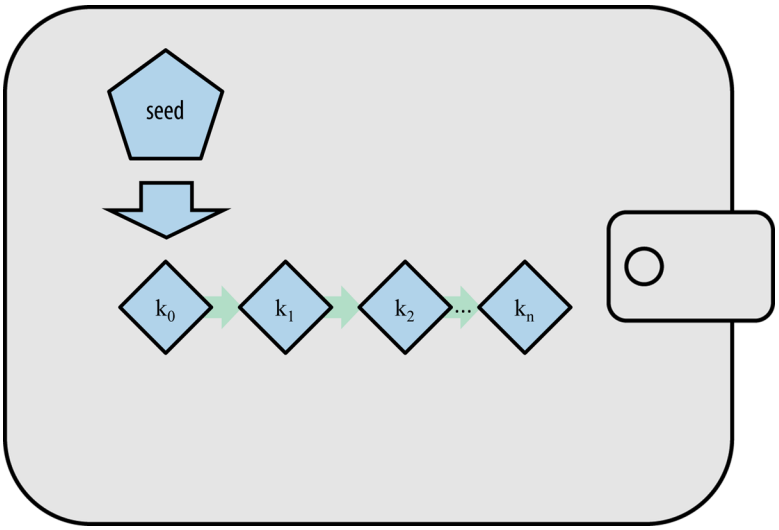


Figura 2 Wallet deterministico. Ogni account k dipende dal precedente. [8]

Nel caso in cui si perda il wallet sarà necessario ricordare/custodire solamente il seed dato che a partire da questi si potranno generare nuovamente tutti gli account.

Il *seed* può essere un valore qualunque. Ad esempio, può essere rappresentato in esadecimale come segue:

```
0C1E24E5917779D297E14D45F14E1A1A
```

2.3.3 Wallet gerarchico (HD Wallet)

Quest'ultima tipologia di wallet si può dire che sia una versione avanzata del wallet deterministico.

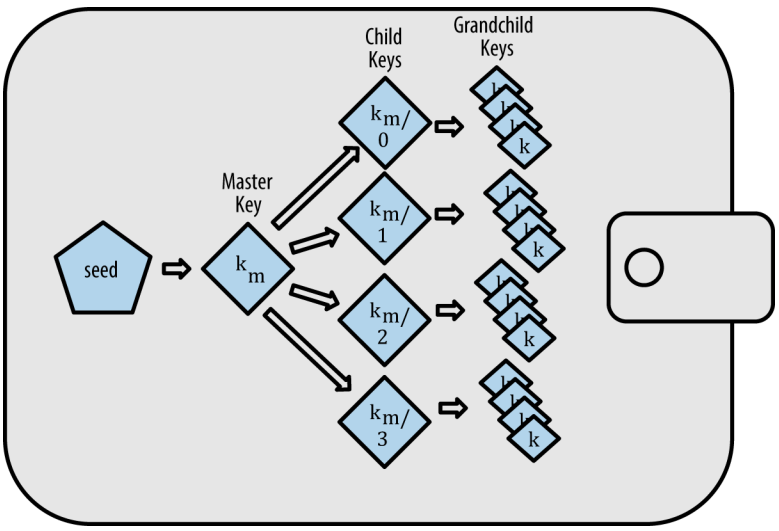


Figura 3 Struttura di un HD Wallet [8]

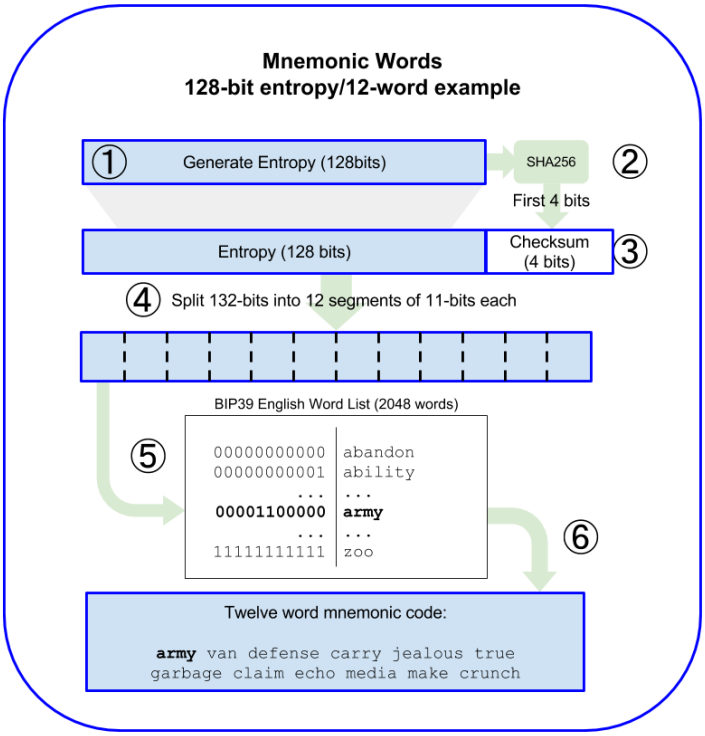
La differenza sostanziale è che gli *accounts* non vengono generati in cascata, bensì ci è permesso di creare una struttura ad albero in modo da generare gli *account* nella posizione a noi preferita.

2.3.4 Generazione del seme

Questo standard permette di rappresentare il *seed* come una serie di parole: una frase mnemonica. La frase mnemonica può essere da 12-24 parole. Il vantaggio di avere una frase mnemonica è che è più facile memorizzarla rispetto ad un *seed* numerico. Ecco un esempio:

```
army van defense carry jealous true
garbage claim echo media make crunch
```

Le parole usate in questo *seed* non sono scelte dall'utente ma sono generate utilizzando un algoritmo definito dallo standard.



1. Crea una sequenza casuale (entropia) da 128 a 256 bit.
2. Crea un checksum della sequenza casuale prendendo i primi bit (entropia-lunghezza/32) del suo hash SHA256.
3. Aggiungi il checksum alla fine della sequenza casuale.
4. Dividere il risultato in segmenti di lunghezza di 11 bit.
5. Associa ogni valore a 11 bit a una parola dal dizionario predefinito di 2048 parole.
6. Il codice mnemonico è la sequenza di parole.

2.4 Ethereum

Ethereum è un'implementazione della tecnologia blockchain. A differenza di bitcoin implementa gli *smart contracts* i quali danno la possibilità di programmare con un linguaggio *general purpose*. Un elemento essenziale di Ethereum è la Ethereum Virtual Machine (EVM), un componente chiave che consente l'esecuzione sicura e decentralizzata degli smart contract all'interno della rete. Gli smart contract, scritti in linguaggi di programmazione ad alto livello come Solidity, vengono compilati in

bytecode prima di essere eseguiti nella EVM. Il *bytecode* sarà poi utile per calcolare il costo delle transazioni.

2.4.1 Smart Contracts

Come già anticipato, la rete Ethereum fornisce il concetto di *smart contract*, ovvero programmi che "vivono" sulla blockchain. Questi vengono scritti tramite alcuni linguaggi, tra i quali spicca Solidity. Una volta fatto il *deploy* sulla rete non sarà più possibile modificare il programma dato che la blockchain è immutabile. Dopo il *deploy* verrà assegnato al codice un *hash* che indentifica il contratto e tramite il quale, associato all'*ABI*, si potrà richiamare le sue funzioni.

L'*ABI* (*Application Binary Interface*) è il modo standard per interagire con i contratti di Ethereum, sia dall'esterno della blockchain che per l'interazione tra contratti. I dati sono codificati in base al loro tipo, come descritto in questa specifica. La codifica non è autodescrittiva e richiede quindi uno schema per essere decodificata. [9]

2.4.2 Gas

Nell'interazione con gli smart contract su Ethereum, il *gas* rappresenta l'unità di misura della complessità computazionale necessaria per eseguire una determinata funzione. Per calcolare il consumo di *gas*, viene assegnato un peso specifico a ciascuna istruzione a basso livello presente nel sistema. In questo modo, è possibile determinare il costo computazionale delle istruzioni contenute nel *bytecode* di una funzione, garantendo un meccanismo equo per valutare l'utilizzo delle risorse nella rete.

Tramite il gas è possibile assegnare un prezzo ad ogni transazione basato sulla difficoltà dell'operazione, facendo pagare una piccola tassa (*fee*) a chi la vuole eseguire. Il gas viene introdotto principalmente per tre motivi:

1. Impedire spam e loop di funzioni: sulla rete Ethereum ogni operazione ha un costo massimo spendibile; se questo viene raggiunto l'operazione viene annullata senza continuare indefinitamente.
2. Incentivare i miners: per garantire la consistenza della rete blockchain sono necessari dei validatori; questi validatori hanno una ricompensa in denaro, la quale viene fornita dalle *fees*.
3. Implementare una priorità: è possibile pagare più ether per il gas; questo consente, nella pratica, che due transazioni identiche costino in modo diverso. In questo modo la transazione pagata di più sarà validata prima.

In sintesi, il gas rappresenta un meccanismo essenziale per garantire l'efficienza, la sicurezza e la priorità delle transazioni sulla rete Ethereum. [10]

3 OAuth2 e OIDC

OAuth 2.0 è un framework (un insieme di regole, linee guida e librerie predefinite) per la delegazione dell'accesso alle API.

A seconda del tipo di flusso, garantisce *access* o *refresh token* che consentono l'accesso alle risorse, anziché immettere username e password, come invece succede tradizionalmente. [11]

Viene quindi creato un sistema ad accesso singolo (SSO, Single Sign-On) che permette, nella pratica, di utilizzare una singola coppia di credenziali per più applicazioni.

3.1 Componenti

Parliamo ora dei principali elementi che compongono il framework OAuth 2.0

3.1.1 Flusso di autorizzazione e attori

Il funzionamento di OAuth è quanto segue.

Ci sono quattro attori fondamentali

- L'utente o r*esource owner*
- Il *resource server* che contiene le risorse al quale l'utente può accedere
- Il *client*
- Il server di autorizzazione o *authorization server*

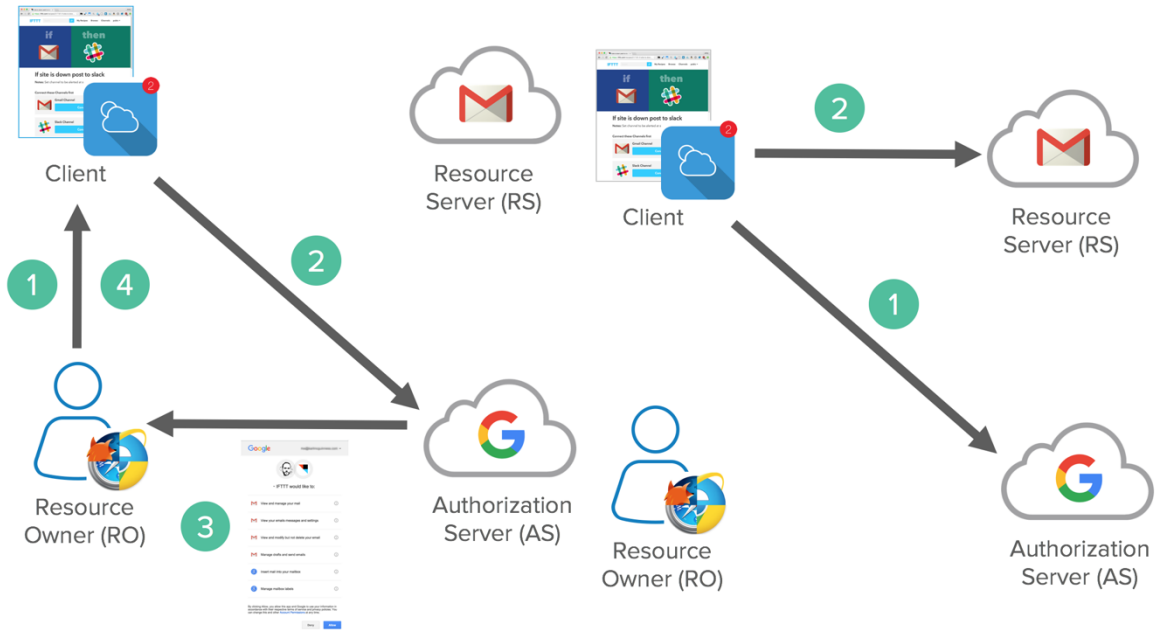


Figura 4 Flusso autorizzazione [11]

Figura 5 Scambio di token [11]

a) Come anche mostrato in Figura 1, il flusso di autorizzazione inizia come segue:
   1. Il *resource owner* delega la gestione dell'accesso al client
   2. Il *client* richiede all'*authorization server* il token d'accesso e gli *scopes* desiderati.
   3. L*'authorization server* comunica all'utente la volontà del *client* di accedere ai dati del *resource owner*. Per procedere l'utente deve accettare
   4. L'autorizzazione viene quindi confermata al client
b) Continua poi, come in Figura 2, lo scambio del token di autentificazione:
   1. Il *client* manda all'*authorization server* il token di accesso chiedendo di utilizzare e/o accedere ad una determinata risorsa
   2. Se il token è valido per quella richiesta, il *resource server* garantirà l'accesso

3.1.2 Scopes

Una delle particolarità di OAuth è la possibilità di utilizzare degli *scopes*. Gli *scopes* sono il gruppo di permessi che il client necessita e che l'utente deve confermare. Ad esempio, se il client ha bisogno di utilizzare nome, cognome e indirizzo e-mail del *resource owner*, chiederà esplicitamente all'utente se consente l'accesso a queste risorse.

Questo succede nel punto a.3 del flusso.

L'utente, di principio, può scegliere se consentire l'accesso a quanti *scopes* desidera. Seguendo l'esempio precedente, potrebbe garantire l'accesso all'email ma non al nome ed al cognome.

Il client si comporterà di conseguenza, obbligando l'utente ad autorizzare certe risorse fondamentali o gestendo l'assenza di alcune.

Nota: è importante ricordare che non si parla solo di accesso a risorse "statiche". Le risorse nel *resource server* possono essere anche azioni, come chiamate ad un determinato endpoint di un API.

3.1.3 Tokens

Impossibile parlare di OAuth2 senza nominare i *tokens*.

I *token* sono dei "gettoni" che il *client* utilizza per accedere al *resource server* o creare il *token* d'accesso. Proprio come i gettoni del parco dei divertimenti, danno un accesso per un periodo limitato ad un servizio.

In questo contesto esistono vari tipi di *token*:

1. *Access token*:
   Questo è il token che il client utilizza per accedere al *resource server*. Ha una durata limitata (minuti o ore) e per questo motivo non può essere revocato. Qualsiasi tipo di *client* può richiedere questo token, anche i *client* non identificato dall'*authorization server*.
2. *Refresh token*:
   Questo è il *token* che il *client* utilizza per richiedere altri *access token*. Non tutti i *client* possono utilizzare questo token, ma solo i *client* identificati ed accettati dall'*authorization server*.
   Questo *token* è a lunga durata (giorni, mesi, anni) e per questo motivo può essere revocato.

Lo standard OAuth non specifica che tipo di *token* usare ma generalmente si utilizzano i *JSON Web Tokens (JWT)*.

JWT è uno standard sicuro ed affidabile per i *token* di autenticazione. *JWT* è una stringa composta da tre sezioni:

1. L'intestazione, *header*
2. Il corpo, *payload*
3. La firma, *signature*

Ogni parte è separata da un punto:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
.
eyJzdWIiOiJ1c2Vycy9Uek1Vb2NNRjRwIiwibmFtZSI6IlJvYmVydCBUb2tlbiBNYW4iLCJzY29
wZSI6InNlbGYgZ3JvdXBzL2FkbWlucyIsImV4cCI6IjEzMDA4MTkzODAif
.
1pVOLQduFWW3muii1LExVBt2TK1-MdRI4QjhKryaDwc
```

L'*header* contiene la tipologia del token (JWT) e la tipologia di algoritmo utilizzato per firmare il token stesso

Il *payload* è il contenuto vero e proprio del messaggio. Nel nostro caso conterrà le informazioni dell'utente che sono condivise tra il *client* ed il *server*.

La *firma* è un meccanismo di sicurezza che garantisce l'integrità e l'autenticità dei dati contenuti nel token. È generata con un algoritmo di firma che utilizza una chiave segreta condivisa tra *client* e *server*, al fine di verificare che i dati non siano stati manomessi durante la trasmissione. [12]

3.2 Open ID Connect (OIDC)

OAuth non era stato concepito per l'autenticazione, ma era stato creato per consentire alle applicazioni di scambiare permessi per particolari risorse. Tuttavia, OAuth come mezzo di autenticazione ha iniziato a diffondersi in numerosi siti web. Da qui è nata la necessità di un protocollo di autenticazione vero e proprio. (OAuth non risponde al "chi è l'utente?" ma al "cosa può fare l'utente?").

Per risolvere questa lacuna, utilizziamo Open ID Connect, il quale ci consente di avere dei *claims* associati all'utente. [13]

3.2.1 Claims

I *claims* sono simili agli *scopes* del protocollo OAuth ma anziché essere generiche autorizzazioni sono vere e proprie informazioni associate all'utente.

In precedenza, infatti era possibile utilizzare lo stesso il protocollo OAuth per dare, ad esempio, l'accesso al nome dell'utente ma questo era trattato come un qualsiasi dato. Invece OIDC associa i dati dell'utente, all'utente stesso così da creare un "pacchetto unico".

Nella pratica i *claims* sono formati dalla coppia "*chiave:informazione*" in formato JSON. I *claims* possono essere standard, come nome, cognome, indirizzo, numero di telefono o indirizzo, oppure personalizzati.

Esempio di utente con alcuni claims standard:

```
{
 "sub": "alice",
 "email": "alice@wonderland.net",
 "email_verified": true,
 "name": "Alice Adams",
 "given_name": "Alice",
 "family_name": "Adams",
 "phone_number": "+359 (99) 100200305",
 "profile": "https://c2id.com/users/alice"
}
```

Nel nostro caso andremo anche ad aggiungere un claim personalizzato contenente un wallet Ethereum ed i suoi account.

3.2.2 Sicurezza

Per garantire la sicurezza del protocollo di autenticazione, OIDC utilizza una serie di tecniche, come l'autorizzazione basata su token, la firma digitale, la crittografia e la validazione del token. In particolare, OIDC utilizza la firma digitale per garantire l'integrità dei token inviati tra il server di autorizzazione e l'applicazione client.

Ogni token emesso dal server di autorizzazione viene firmato digitalmente utilizzando una chiave privata, che può essere verificata dall'applicazione client utilizzando la chiave pubblica corrispondente. Questo garantisce l'integrità dei token inviati tra il server di autorizzazione e l'applicazione client, impedendo a eventuali aggressori di manipolare o contraffare i token.

4 Tecnologie usate

Qui sono elencate le tecnologie che sono state utilizzate per lo sviluppo della mia soluzione divise in front-end e back-end.

Come già anticipato, l'obiettivo è sviluppare una soluzione che permetta all'utente l'accesso alla blockchain, ed eventuali funzioni che sfruttano la tecnologia, tramite l'uso di credenziali. Si tenta quindi di limitare il più possibile l'interazione con elementi della Blockchain che possono essere estranei all'utente comune.

4.1 Front-End

4.1.1 Node.js

Node.js è una piattaforma potente e popolare per costruire applicazioni di rete veloci e scalabili che è stata scelta per il progetto.

I vantaggi portati dall'utilizzo di Node.js sono molteplici. Esso è creato per essere usato in modo asincrono e non bloccante. Questo consente di avere applicazioni veloci e reattive. Inoltre, consente di scalare il software senza avere perdite di prestazioni.

L'utilizzo di Node.js è, inoltre, molto comune: questo consente di avere un'ampia community e vari siti sul quale trovare informazioni in caso di necessità.

Degno di nota è anche la sua compatibilità *multiplatform* che consente quindi di poter usare Node.js su tutti i dispositivi senza problemi.

Infine, ultimo ma non per importanza, dà la possibilità di scaricare le librerie dalle repository pubbliche ed in modo automatico: è possibile quindi definire un set di *imports* che verranno scaricati automaticamente ed aggiornati se previsto. [14]

4.1.2 Vue.js

Vue.js è un popolare framework JavaScript front-end che consente agli sviluppatori di costruire pagine *single-page*. Ecco alcune delle sue principali caratteristiche e vantaggi: [15]

- Reattivo e componibile: Il suo sistema di reattività assicura che le modifiche ai dati si riflettano immediatamente nell'interfaccia utente, mentre la sua architettura basata su componenti consente agli sviluppatori di costruire componenti riutilizzabili e modulari.
- Leggero e veloce: Il DOM virtuale assicura che le modifiche ai dati siano elaborate e renderizzate in modo efficiente, il che può tradursi in tempi di caricamento delle pagine più rapidi e in un'esperienza utente più fluida.

- Versatile e flessibile: Può essere facilmente integrato con altre librerie e framework e può essere utilizzato con vari strumenti e tecnologie front-end.

4.1.2.1 Vuex

Vuex è una libreria di gestione dello stato per le applicazioni web basate su Vue.js. Lo stato è l'insieme dei dati che rappresentano lo stato dell'applicazione in un dato momento, come ad esempio i dati dell'utente, i dati di configurazione, i dati di cache, ecc.

Tutti i dati dell'applicazione sono conservati in un singolo store, che viene gestito attraverso una serie di mutazioni, azioni e getter.

4.1.2.2 Vue Router

Vue Router è una libreria di *routing* per le applicazioni web basate su Vue.js. Il *routing* è il processo attraverso il quale l'applicazione gestisce la navigazione dell'utente tra le diverse pagine o sezioni dell'applicazione. Vue Router fornisce un'interfaccia per gestire il *routing* dell'applicazione in modo semplice.

Vue Router funziona attraverso la definizione di diverse *route*, o percorsi, che l'applicazione può gestire. Una *route* è definita da una coppia di valori: un *path*, che rappresenta l'URL della *route*, e un componente, che rappresenta il contenuto della pagina corrispondente all'URL.

Permette anche la gestione dei parametri degli URL e la gestione delle animazioni di transizione tra le pagine.

4.1.3 Web3.js

Web3.js è una libreria JavaScript che fornisce una raccolta di funzioni e API per interagire con la blockchain di Ethereum.

Essa fornisce un modo semplice e conveniente per connettersi alla blockchain, che sia locale o remota, utilizzando vari protocolli, tra cui HTTP, WebSockets.

Inoltre, permette l'interazione con gli smart contract sulla blockchain, come l'invio di transazioni, la chiamata di metodi di contratto o l'interrogazione dello stato del contratto, creazione di nuovi account, l'importazione di account esistenti e la firma delle transazioni.

Fornisce, infine, la possibilità di ascoltare gli eventi emessi dalla blockchain, come nuovi blocchi o eventi degli smart contracts. Questo può essere utile per costruire applicazioni in tempo reale che devono rispondere ai cambiamenti sulla blockchain.

Può essere facilmente integrato con framework front-end, come Vue.js. Ciò consente di creare interfacce che interagiscono con la blockchain.

4.1.4 Typescript

TypeScript è un linguaggio di programmazione open source creato da Microsoft, che si pone come un'evoluzione di JavaScript.

TypeScript permette l'utilizzo dei tipi in JavaScript. Ciò permette agli sviluppatori di definire i tipi di variabili, parametri e valori di ritorno delle funzioni durante la stesura del codice, aumentando l'affidabilità e facilitando la manutenzione del codice stesso.

Oltre a questo, propone una serie di funzionalità avanzate, tra cui la gestione delle classi, l'ereditarietà, le interfacce e i moduli, rendendolo particolarmente adatto a progetti di ampia portata e allo sviluppo di applicazioni web attuali.

Utilizzare TypeScript offre numerosi vantaggi rispetto al solo JavaScript. Prima di tutto, la tipizzazione statica consente di identificare errori di tipo durante la compilazione piuttosto che durante l'esecuzione, minimizzando il rischio di errori di programmazione.

4.2 Back-End

4.2.1 Jakarta Persistence

Jakarta Persistence è una delle API che fanno parte di Jakarta EE. Questa API consente di gestire in modo efficiente e semplificato la persistenza dei dati nelle applicazioni Java, in particolare nelle applicazioni aziendali. La persistenza dei dati si riferisce al processo di memorizzazione delle informazioni in modo che possano essere recuperate e utilizzate in seguito, anche dopo che il sistema è stato spento o riavviato.

I suoi punti fondamentali sono:

1. Object-Relational Mapping e operazioni CRUD:
   - Meccanismo di ORM fornito da Jakarta Persistence per mappare le classi Java agli schemi di tabelle di database relazionali
   - Possibilità di eseguire operazioni CRUD (Create, Read, Update e Delete) sui dati memorizzati nel database attraverso un'interfaccia ad alto livello e orientata agli oggetti, senza dover scrivere direttamente le query SQL
   - Utilizzo di annotazioni Java per definire le entità e mapparle nelle tabelle del database
2. EntityManager:
   - Descrizione del concetto di "Entity" come classe Java che rappresenta una tabella nel database e le sue istanze come righe della tabella
   - Ruolo dell'EntityManager nella gestione del ciclo di vita delle entità e delle transazioni con il database

- Possibilità di eseguire operazioni come creare, leggere, aggiornare e cancellare entità, gestire le transazioni e sincronizzare lo stato delle entità con il database
- Possibilità di ottenere l'EntityManager tramite l'iniezione di dipendenza o attraverso una *factory*, a seconda delle esigenze dell'applicazione

3. JPQL:
   - È un linguaggio di interrogazione simile a SQL ma orientato agli oggetti
   - Serve per eseguire query sulle entità e le loro relazioni in modo più intuitivo rispetto al SQL standard
   - Utilizzato quando si lavora con modelli di dati complessi che coinvolgono relazioni tra diverse entità.

4.2.2 Java Spring

Java Spring è un popolare framework open-source per la creazione di applicazioni Java. [16]

Caratteristico è il supporto alla iniezione di dipendenze, che consente di scrivere codice facilmente testabile e strutturato. Le dipendenze vengono iniettate in una classe anziché essere create all'interno della classe stessa.

Fornisce un supporto per lo sviluppo web, compresa la creazione di servizi web RESTful, applicazioni web MVC e applicazioni basate su WebSocket.

Infine, fornisce alcune classi per creazioni comuni, come quella di un Authorization Server.

5 Progettazione

Passiamo ora all'idea vera e propria.

Il nostro scopo è quindi di salvare un *wallet* crittografico in un *server* utilizzando *OIDC* e mantenendo una sicurezza adeguata.

Come già presentato in precedenza, *OIDC* consente di aggiungere dei *claims* personalizzati, in accordo con i principi di autenticazione.

L'idea è quindi di associare il *wallet* all'account inserendolo in un *claim* creato appositamente.

5.1 Interazione con la dApp

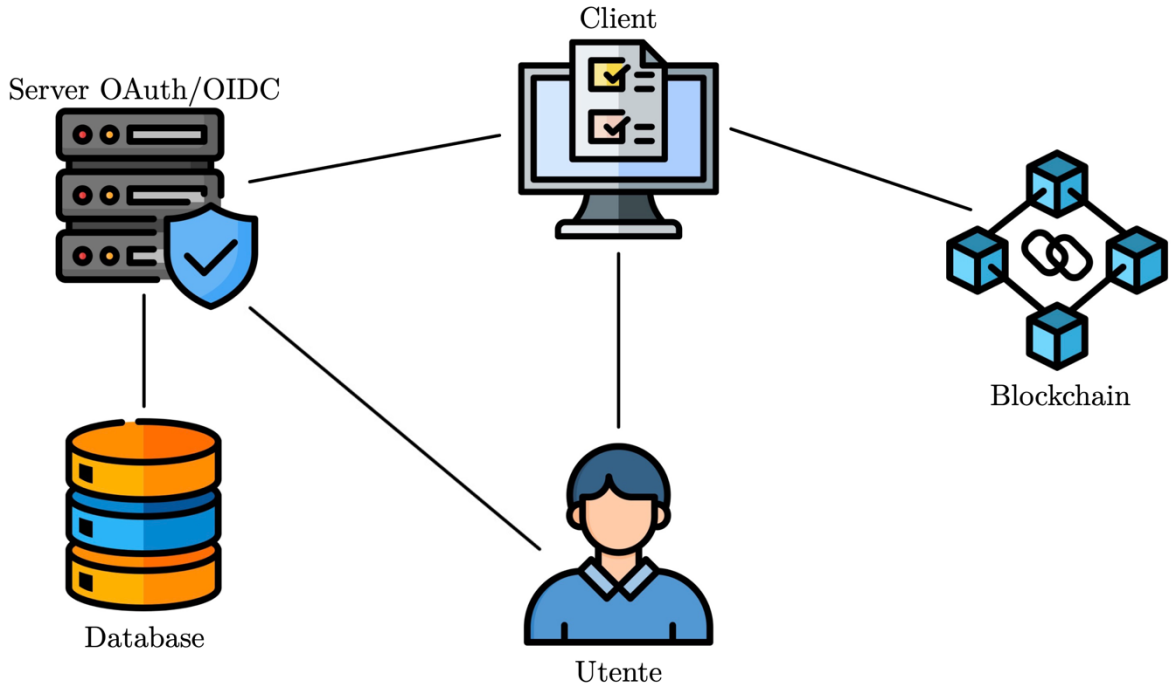


Figura 6 Interazioni tra le parti

L'utente si interfaccia, via *browser*, con il *client* per fare l'accesso. A questo punto il *client* reindirizza l'utente sulla pagina di accesso del *server* OAuth.

La pagina di accesso chiede all'utente di inserire e-mail e password. Successivamente chiede all'utente se il *client* può accedere alle informazioni come nome, cognome e *wallet*. Se l'utente accetta, il *server* OAuth recupera i dati dal *database* e li inoltra al *client*.

L'utente viene quindi reindirizzato alla schermata principale del *client*.

I dati del wallet sono però cifrati, quindi potrebbe essere necessaria un'ulteriore interazione da parte dell'utente. Nel momento in cui l'utente tenterà di compiere un'operazione di scrittura, o che in qualche modo richieda l'accesso alla chiave privata di un *account* del *wallet*, verrà chiesto di inserire la password associata. Si potrà quindi utilizzare la funzione interessata.

5.2 Salvataggio dell'utente

Per mantenere le informazioni degli utenti in modo persistente, sarà necessario memorizzarle in un database. In questo caso, verrà scelto di utilizzare un database SQL e la libreria Jakarta per connettersi ad esso.

Per rappresentare gli utenti all'interno del database, verrà creata una classe apposita. Questa classe avrà diversi attributi che rappresenteranno le informazioni dell'utente, come l'e-mail, la password, il nome, il cognome e altri dettagli.

Inoltre, la classe utilizzerà alcune annotazioni di JPA (Java Persistence API), come @Entity e @Table, per indicare che la classe rappresenterà una entità persistente e per specificare il nome della tabella del database associata alla classe.

Per mantenere l'utente in memoria sarà necessaria una forma di persistenza. Si deciderà quindi di utilizzare un database SQL e la libreria Jakarta per connettersi ad esso.

5.3 Salvataggio del wallet

Le librerie web3.js forniscono già uno standard per il *wallet*, sia se si parla di *wallet* non cifrato che di *wallet* cifrato.

Il *wallet plain* è proposto in sottoforma di *json* come segue:

```
{
  0: {...}, // account by index
  "0xF0109fC8DF283027b6285cc889F5aA624EaC1F55": {...},  // same account
                                                        // by address
  "0xf0109fc8df283027b6285cc889f5aa624eac1f55": {...},  // same account
                                                       // by address lowercase
  1: {...},
  "0xD0122fC8DF283027b6285cc889F5aA624EaC1d23": {...},
  "0xd0122fc8df283027b6285cc889f5aa624eac1d23": {...},
  / ...funzioni... /
  length: 2,
}
```

Dove ad ogni indirizzo corrisponde un account:

```
{
  address: "0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa01",
  privateKey: "0x348ce564d427a3311b6536bbcff9390d69395b06ed6c48695...",
  / ...funzioni... /
  encrypt: function(password){...}
}
```

La libreria propone sia una cifratura per *account*, sia una cifratura per *wallet*. Per la dimostrazione si è deciso di cifrare l'intero *wallet*. Per una maggiore efficienza è necessario cifrare e decifrare il singolo account.

Il *wallet* cifrato verrà formato come segue:

```
[{
  version: 3,
  id: 'dcf8ab05-a314-4e37-b972-bf9b86f91372',
  address: '06f702337909c06c82b09b7a22f0a2f0855d1f68',
  crypto:
    {
      ciphertext: '0de804dc63940820f6b3334e5a4bfc8214e27fb30bb7e9b7...',
      cipherparams: [Object],
      cipher: 'aes-128-ctr',
      kdf: 'scrypt',
      kdfparams: [Object],
      mac: 'b2aac1485bd6ee1928665642bf8eae9ddfbc039c3a673658933d320b...'
    }
},
  {
    version: 3,
    id: '9e1c7d24-b919-4428-b10e-0f3ef79f7cf0',
    address: 'b5d89661b59a9af0b34f58d19138baa2de48baaf',
    crypto:
      {
        ciphertext: 'd705ebed2a136d9e4db7e5ae70ed1f69d6a57370d5fbe0...',
        cipherparams: [Object],
        cipher: 'aes-128-ctr',
        kdf: 'scrypt',
        kdfparams: [Object],
        mac: 'af9eca5eb01b0f70e909f824f0e7cdb90c350a802f04a9f6afe056...'
      }
  }
]
```

Questo testo *json* è ciò che verrà poi salvate nel *claim* personalizzato di dell'*OIDC*.

5.4 Gestione della password

Anche se l'utente utilizzerà una sola *password,* in realtà, le *password* utilizzate a livello più basso sono due:

- *password* del profilo *OIDC*
- *password* del *wallet* cifrato contenuto nel *claim*

Queste *password*, di principio, potrebbero anche essere differenti: questo consentirebbe di avere una "doppia sicurezza": per usufruire della chiave privata degli *account* Ethereum, si dovrebbe inserire una *password* differente da quella del profilo.

Dato che l'obbiettivo di questo progetto è, però, anche rendere fruibile la tecnologia ad un utente non esperto, si è scelto di "sincronizzare" le password, in modo da usare sempre la stessa.

Per chiarezza, si riporta un esempio di scenario di successo:

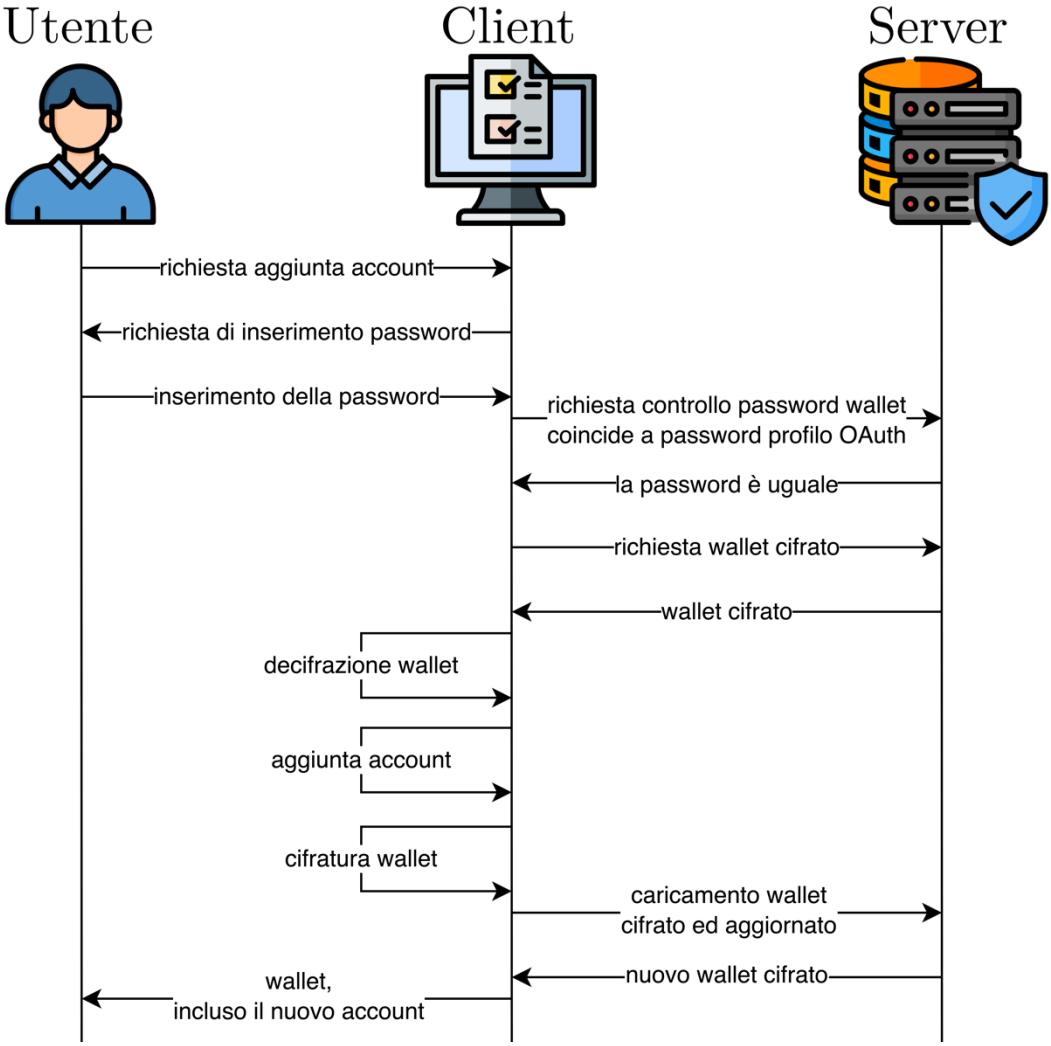


Figura 6 Richiesta aggiunta account, scenario di successo.

In questo scenario si suppone di star aggiungendo un *account* al *wallet* associato al mio profilo OAuth. Questa operazione viene eseguita dopo aver eseguito con successo l'accesso alla piattaforma.

Quando l'utente richiede di aggiungere un *account*, tramite l'inserimento della chiave privata o tramite la frase mnemonica, viene richiesto l'inserimento della *password*.

Una volta inserita la *password*, l'*hash* viene mandato al *server* che risponderà in modo affermativo, dato che parliamo di uno scenario di successo.

A questo punto il *client* chiede al *server* il *wallet*, il *server* risponde con il *wallet* in forma cifrata, ed il *client* aggiunge l'*account*, decifrando e cifrando in maniera opportuna.

Infine, il *wallet* aggiornato sarà mandato sul *server* che, una volta salvato, risponderà con il *wallet* aggiornato appena aggiunto, che verrà mostrato all'utente.

6 Implementazione

Passiamo ora a come ho sviluppato la soluzione.

6.1 Server-side

6.1.1 Configurazione Authorization Server

Per lo sviluppo del server ho usato il framework Java Spring che ci consente di utilizzare alcune classi che predispongono alla creazione del server.

Guardiamo ora le parti fondamentali del codice.

Inizio configurando le informazioni del client che si collegherà al server e quali tipo di servizi OAuth il server esporrà.

```
@Bean
public RegisteredClientRepository registeredClientRepository(
    PasswordEncoder passwordEncoder
) {
  RegisteredClient registeredClient = RegisteredClient.withId(
          UUID.randomUUID().toString())
      .clientId("crypto-client")
      .clientSecret(passwordEncoder.encode("secret"))
      .clientAuthenticationMethod(
          ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
      .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
      .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
      .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
      .redirectUri("http://127.0.0.1:5173/callback")
      .scope(OidcScopes.OPENID)
      .scope(OidcScopes.PROFILE)
      .scope(OidcScopes.EMAIL)
      .scope("wallet")
      .clientSettings(ClientSettings.builder()
          .requireAuthorizationConsent(true).build()
      ).build();
  return new InMemoryRegisteredClientRepository(registeredClient);
}
```

Qui definisco le informazioni:

- *client id*: che si connetterà
- *client secret*: “password” condivisa tra client e server
- *metodo di autenticazione*
- *tipo di concessione*: tipologia di token utilizzabili
- *redirect URI*: URI che il client dovrà obbligatoriamente chiamare una volta terminato l’accesso
- *scope:* scopes che il server fornisce

Qui, dovendo solo interfacciarci con una tipologia di client, creiamo una configurazione statica. Nulla ci vieta però di dare la possibilità di registrare più clients tramite un’interfaccia e caricare gli stessi dal database.

6.1.2 Database

Per mantenere i dati degli utenti in memoria è stato necessario interfacciarsi con un piccolo database.

Per la creazione e la gestione uso la libreria Jakarta che consente di connettersi ad un database vuoto, strutturarlo e popolarlo tramite Java.

Questa è la configurazione del file yaml di Spring per l'interazione con il database:

```yaml
spring:
  jpa:
    hibernate:
      ddl-auto: update
  datasource:
    url: jdbc:mysql://localhost:8889/oauth2
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
```

Dopodiché definiamo un "utente database" come segue:

```java
@Entity
@Table(name = "users")
public class CryptoUserDatabase {

  @Id
  @Column(nullable = false)
  private String email;

  @Column(nullable = false)
  private String password;

  @Column
  private String firstName;

  @Column
  private String lastName;

  @ElementCollection(fetch = FetchType.EAGER)
  @CollectionTable(name = "authorities")
  private List<String> grantedAuthorities = new ArrayList<>();

  @Column(columnDefinition = "text")
  private String wallet = "[[],[],[]]";

  @Column(nullable = false)
  private boolean isAccountExpired = false;

  @Column(nullable = false)
  private boolean isAccountLocked = false;

  @Column(nullable = false)
  private boolean isCredentialsExpired = false;

  @Column(nullable = false)
  private boolean isEnabled = true;
  /* ... */
}
```

Come si può notare, l'utente sul database contiene tutte le informazioni che vogliamo e dobbiamo memorizzare. Alcune sono date dallo standard degli utenti che usa Spring, come *isEnabled*, altre create ad hoc per il progetto, come *wallet*.

Abbiamo bisogno poi di due sottoclassi di utenti:

- "Utenti Spring": implementano la classe *UserDetails* di Spring e consente di gestire gli accessi diretti al server.
- Utenti OIDC: estendono la classe *OidcUserInfo* e permettono di utilizzare le funzionalità utente di *Open ID Connect*.

Anche se all'esterno sono utilizzati per situazioni diverse, l'utente che viene recuperato è sempre lo stesso. La differenza è come viene formato il suo "involucro".

Per gestire gli utenti e le modifiche si è creata una classe *CryptoUserDetailsService* che si interfaccia con entrambe le tipologie di utente.

Di seguito i due metodi che richiamano il database

```
@Override
public UserDetails loadUserByUsername(String username) {
  CryptoUserDatabase cryptoUserDatabase =
                                  userRepository.findByEmail(username);
  if (cryptoUserDatabase == null) {
    throw new UsernameNotFoundException(username);
  }
  return new CryptoUserPrincipal(cryptoUserDatabase);
}

public OidcUserInfo loadOidcUserByUsername(
  String username,
  Set<String> scopes) {
  CryptoUserDatabase cryptoUserDatabase =
                                  userRepository.findByEmail(username);
  if (cryptoUserDatabase == null) {
    throw new UsernameNotFoundException(username);
  }
  return new CryptoUserOidc(cryptoUserDatabase, scopes);
}
```

Oltre a questi ci sono alcuni metodi per la gestione della modifica dell'utente:

```
public void addCryptoUser(CryptoUserDatabase user);
public void updateCryptoUserWallet(String username, String wallet);
public void updateCryptoUserPassword(String username,
                                     String newEncodedPassword);
```

Che servono rispettivamente per:

- Aggiungere un utente al database
- Aggiungere/aggiornare il wallet
- Aggiungere/aggiornare la password

Per motivi dimostrativi non vengono implementate altre funzioni.

Infine, per unire la classe utente con il database vero e proprio, così da fornire tutte le funzioni utili viene creata questa interfaccia:

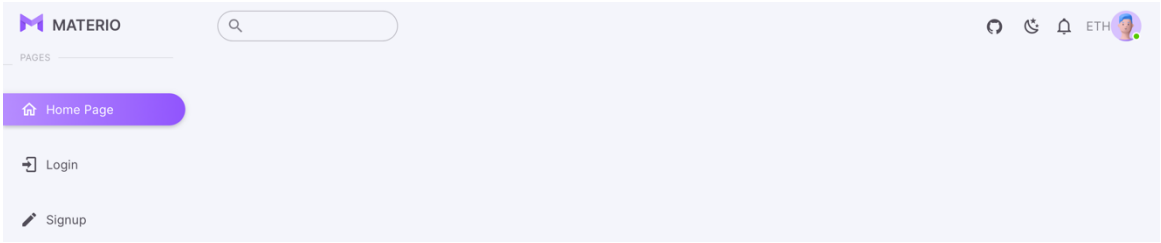
```
public interface UserRepository extends
                                JpaRepository<CryptoUserDatabase, Long> {
  CryptoUserDatabase findByEmail(String username);
}
```

Si può notare che l'interfaccia viene usata nella classe *CryptoUserDetailsService* per recuperare l'utente e castarlo.

6.2 Client-side

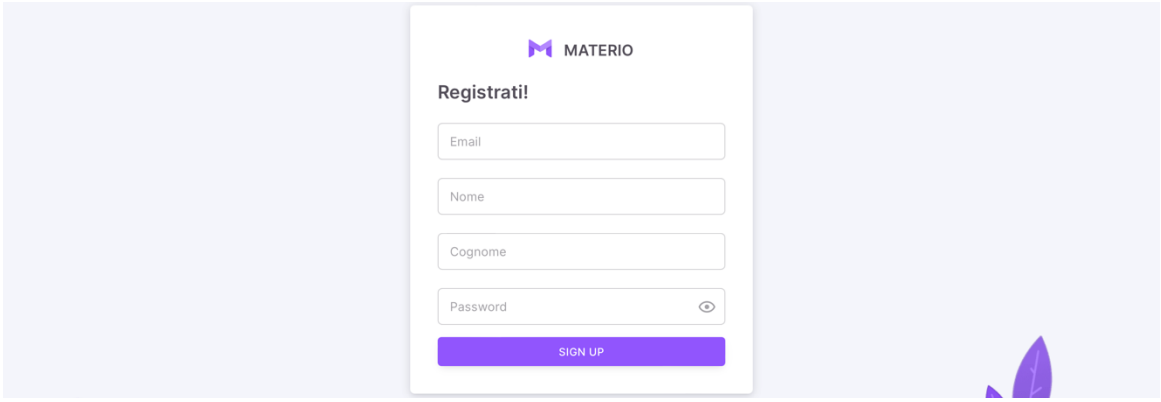
6.2.1 Quadro generale

Al momento del primo accesso la pagina si mostra come segue:



Viene mostrato quindi la pagina home dell'utente *non loggato*, da implementare a piacere, e viene mostrato il tasto di *login* e *signup*.

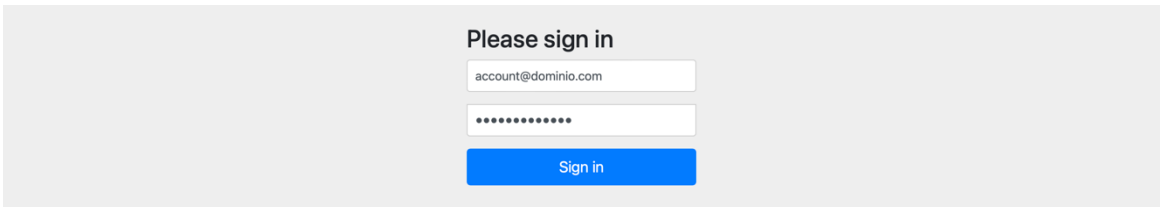
6.2.2 Signup



Attraverso la pagina di signup sarà possibile creare un nuovo utente per poter successivamente accedere alle funzionalità dell'applicazione.

6.2.3 Login

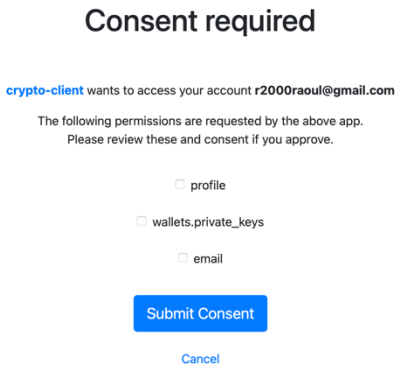
Premendo sul tasto di login si potrà arrivare alla pagina di accesso nella quale è possibile accedere.



Quando viene premuto “Sign in” viene innescato il flusso di autorizzazione che ci consente di accedere. Questa pagina di accesso non è situata nel *client* ma nel *server*, il quale comunicherà poi con il client il *token*.

6.2.4 Scopes

Viene poi mostrata la pagina per consentire gli accessi ai diversi scopes:



L’utente può quindi selezionare i singoli dati che vuole condividere con il client. Per scopi dimostrativi daremo l’accesso a tutti i campi.

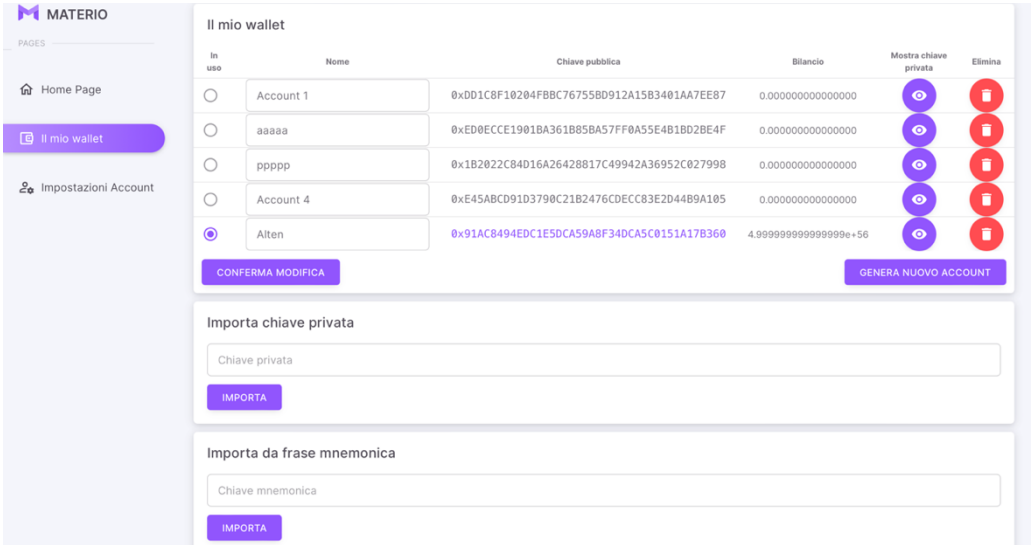
Sarà così possibile vedere le due pagine “Il mio wallet” e “Impostazioni Account”

6.2.5 Il mio wallet

La pagina si presenta divisa in tre sezioni:

- Il mio wallet: questa parte comprende gli wallet attualmente associati all’account; è possibile selezionare il wallet da usare per tutte le funzioni del sito e modificare in nome. Inoltre, è possibile visualizzare e/o eliminare la chiave privata dell’account, dopo un secondo inserimento della password
- Importa la chiave privata: inserendo una chiave privata nell’apposito campo, è possibile associarla al wallet
- Importa da frase mnemonica: inserendo una chiave mnemonica nell’apposito campo, è possibile associarla al wallet. Da notare che viene automaticamente inserito il primo wallet associato alla chiave mnemonica.

Quando si carica questa pagina il wallet viene scaricato cifrato. Infatti, non è necessario decifrare il wallet per vedere la chiave pubblica.
Quando viene fatta una richiesta di modifica o viene richiesto di mostrare la chiave privata verrà decifrato il wallet tramite l’inserimento della password.



Queste sono le tre funzioni utilizzate per prendere il wallet cifrato e decifrato:

```
async getUserInfo(): Promise<IdTokenClaims | undefined> {
  try {
    let user = await userManager.getUser()
    if (user) {
      axios.defaults.headers.common['Authorization'] =
                                  `${user.token_type} ${user.access_token}`
      return user.profile
    } else
      return undefined
  } catch (e) {
    console.error("errror while getting user info:", e)
  }
}
async retrieveEncryptedWallet(): Promise<EncryptedKeystoreV3Json[]> {
  const user = await this.getUserInfo()
  if (user) {
    return <EncryptedKeystoreV3Json[]>
                                    JSON.parse(<string>user['wallet'])[0]
  } else {
    return []
  }
}
async retrieveDecryptedWallet(password: string): Promise<WalletBase> {
  const user: IdTokenClaims = <IdTokenClaims> await this.getUserInfo()
  const encryptedWallet: EncryptedKeystoreV3Json[] =
                                      await this.retrieveEncryptedWallet()

  let passwordMatches: boolean = false
  if (user) {
    passwordMatches = comparePasswordHash(password,
                                          <string>user["password"])
  }
  if (!passwordMatches) {
    throw "La password non corrisponde con quella dell'account"
  }
  return web3.eth.accounts.wallet.decrypt(encryptedWallet, password)
}
```

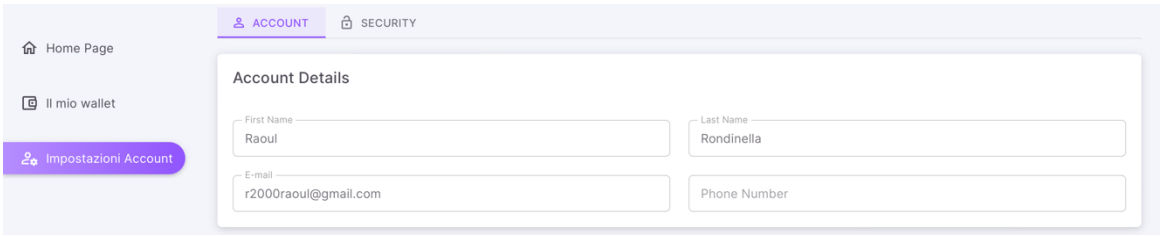
Ho creato due funzioni distinte per recuperare il wallet cifrato e decifrato, in modo da non dover inserire la password anche solo per l'inserimento.

6.2.6 Impostazioni account

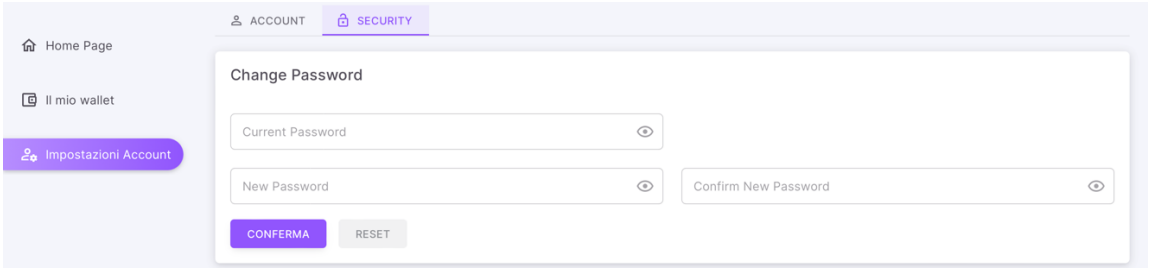
Accedendo alle impostazioni dell'account si presentano due schede:

- Account
- Sicurezza

La scheda *account* mostra le informazioni personali dell'utente quali e-mail, nome, cognome. Non viene visualizzato il wallet in quanto c'è la pagina dedicata "Il mio wallet".



La scheda *sicurezza* mostra un form per il cambio della password utente. Quando viene cambiata la password, viene in realtà cambiata sia la password dell'account sia la chiave di cifratura del wallet. Come anticipato sopra, è stato scelto di lasciare coordinate le due password.



Di seguito, la funzione in Typescript per cambiare la password.

```
async changePassword(currentPassword: string, newPassword: string) {
  const user: IdTokenClaims = await this.signinSilent()
  if (user) {
    if (!comparePasswordHash(currentPassword, <string>user.password)) {
      throw ('Wrong password.')
    }
  }
  try {
    let decryptedWallet = await
                          this.retrieveDecryptedWallet(currentPassword)
    let names = await this.retrieveNames()
    await this.updateRemoteWallet(decryptedWallet, names, newPassword)
    const salt = await bcrypt.genSalt(10)
    const hashedPassword = await bcrypt.hash(newPassword, salt)

    const formData = new FormData()
    formData.append('newPassword', hashedPassword)

    await this.signinSilent()
    await axios.post('http://127.0.0.1:9000/changePassword', formData)
  } catch (error: any) { /* gestione errori */ }
}
```

Conclusioni

Concludendo, credo che sia stato possibile trovare un buon compromesso tra sicurezza e facilità di utilizzo. Questa implementazione è una buona fondamento per la costruzione dell'intero ecosistema presentando un'interfaccia intuitiva, per l'utente e facilmente espandibile, per lo sviluppatore.

L'integrazione di OAuth 2.0 con Ethereum ha permesso di fornire un'autenticazione sicura e semplificata, facilitando la creazione e l'importazione del wallet per gli utenti.

In definitiva, il progetto presentato rappresenta un passo importante verso un utilizzo più ampio della tecnologia blockchain, abbattendo le barriere tecniche e rendendo il processo più intuitivo per gli utenti comuni. Il risultato è una soluzione che permette un accesso sicuro e user-friendly.

Ringraziamenti

Innanzitutto, ringrazio il mio relatore Claudio Schifanella ed il mio tutor di stage Alberto Ferrini per aver reso possibile questo elaborato.

Grazie ai miei genitori, per avermi supportato, e sopportato, durante questi anni di alti e bassi (e pandemie mondiali).

Ai miei ex-professori dell'I.T.T. Amaldi-Sraffa di Orbassano, per avermi costretto a cambiare più volte le mie idee ed avermi dato le basi per l'Università.

Ai miei amici di sempre, Francesco M. e Camilla D., che mi hanno accompagnato durante gli anni con *memes* ed uscite.

Ed ultimo, ma non di certo per importanza, ad Andrea B.: collega, compagno di Erasmus e soprattutto amico, di quegli amici che si fanno un po' fatica a trovare.

Ci siamo insegnati tante cose l'un con l'altro, da quando ci conosciamo, ma un principio mi è rimasto impresso: che nella vita, alle volte, *bisogna spingere*.

Bibliografia

[1] «Alten Italia,» [Online]. Available: https://www.alten.it/chi-siamo-meet-us/. [Consultato il giorno 25 03 2023].

[2] E. Sandrone, «Realizzazione di una piattaforma per l'analisi del flusso di dati in una blockchain Ethereum,» Università degli studi di Torino, Torino, 2020.

[3] «Blockchain: cos'è, come funziona e gli ambiti applicativi in Italia,» 28 09 2022. [Online]. Available: https://www.blockchain4innovation.it/esperti/blockchain-perche-e-cosi-importante/.

[4] «Consensus Mechanisms in Blockchain,» 12 05 2022. [Online]. Available: https://crypto.com/university/consensus-mechanisms-in-blockchain.

[5] «Ethereum - Proof of Work (PoW),» 5 12 2022. [Online]. Available: https://ethereum.org/it/developers/docs/consensus-mechanisms/pow/.

[6] «Ethereum - Proof of Stake (PoS),» 5 12 2022. [Online]. Available: https://ethereum.org/it/developers/docs/consensus-mechanisms/pos/.

[7] «Proof of Authority Explained,» 1 02 2023. [Online]. Available: https://academy.binance.com/en/articles/proof-of-authority-explained.

[8] «BitcoinBook: Wallets,» 07 01 2022. [Online]. Available: https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch05.asciidoc.

[9] «Solidity,» [Online]. Available: https://docs.soliditylang.org/en/v0.8.1/. [Consultato il giorno 07 03 2023].

[10] «Ethereum - Gas and fees,» 17 02 2023. [Online]. Available: https://ethereum.org/en/developers/docs/gas/.

[11] «What the Heck is OAuth?,» 21 06 2017. [Online]. Available: https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth.

[12] «A Beginner's Guide to JWTs in Java,» 21 06 2016. [Online]. Available: https://stormpath.com/blog/beginners-guide-jwts-in-java.

[13] «OpenID Connect,» [Online]. Available: https://portswigger.net/web-security/oauth/openid. [Consultato il giorno 06 03 2023].

[14] «Node.js,» [Online]. Available: https://nodejs.org/en/. [Consultato il giorno 14 03 2023].

[15] «Vue.js,» [Online]. Available: https://vuejs.org. [Consultato il giorno 14 03 2023].

[16] «Spring,» [Online]. Available: https://spring.io. [Consultato il giorno 14 03 2023].

[17] «Oracle Documentation - OAuth 2.0,» [Online]. Available: https://docs.oracle.com/cd/E82085_01/160030/JOS%20Implementation%20Guide/Output/oauth.htm. [Consultato il giorno 2023 02 13].