



Università degli Studi di Torino
Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

**Progettazione e realizzazione di un
exchange decentralizzato per lo
scambio di token ERC20 per la
piattaforma CommonsHood**

Relatore:
Prof. Claudio Schifanella

Candidato:
Shao Hao Hu

Sessione Novembre 2021
Anno Accademico 2020-2021

Abstract

CommonsHood è un'applicazione web basata su smart contract per blockchain Ethereum che ha lo scopo di fornire alla comunità strumenti per l'inclusione finanziaria e per supportare l'economia locale delle comunità di cittadini. L'applicazione permette agli utenti di creare nuovi tipi di token crittografici che possono rappresentare beni di valore o servizi. Tali token, quindi, possono essere utilizzati per creare campagne di crowdfunding, coupons e per altre funzionalità incluse all'interno di CommonsHood. È perciò necessario permettere agli utenti la compravendita e lo scambio dei token creati e distribuiti. Lo scopo principale di questa tesi è l'implementazione delle funzionalità di vendita e acquisto dei token crittografici attraverso la progettazione e realizzazione di un exchange decentralizzato. Nella fase iniziale, attraverso l'uso del linguaggio Solidity, sono stati scritti gli smart contracts che definiscono le regole e il funzionamento delle compravendite (gestite attraverso il meccanismo di domanda e offerta), e, con l'uso del linguaggio Javascript, sono stati realizzati i test che ne verificano il corretto funzionamento. Nella fase successiva è stata creata l'interfaccia web che permette l'utilizzo delle funzioni di compravendita. Questa è stata realizzata ancora tramite il linguaggio Javascript ed utilizzando il framework ReactJS e i componenti grafici inclusi nel framework Material-UI. Per collegare l'applicazione con la blockchain è stato utilizzato il framework web3.js.

Indice

1	Introduzione	7
1.1	CommonsHood	7
1.2	Blockchain	7
1.2.1	I nodi	8
1.2.2	Le caratteristiche di una blockchain	9
1.2.3	Smart contracts	10
1.3	Ethereum	10
1.3.1	Token Ethereum	10
1.3.1.1	Token ERC-20	11
1.3.1.2	NFT	11
1.3.2	Smart contracts in Ethereum	12
1.4	OpenZeppelin	15
1.5	Metamask	15
1.6	ReactJS	15
1.6.1	Caratteristiche di ReactJS	15
1.6.1.1	Componenti	16
1.6.1.2	Hook state	17
1.6.1.3	Hook effect	18
1.7	Material-UI	19
1.8	Truffle e Ganache	19
1.8.1	Creazione di un progetto Truffle	20
1.8.2	Compilazione degli smart contracts	20
1.8.3	Deploy degli smart contracts	21
1.8.4	Test degli smart contracts	22

2	Progettazione	24
3	Scrittura e test degli smart contracts	28
3.1	Scrittura degli smart contracts	28
3.1.1	Smart Contract SaleFactory	28
3.1.1.1	Variabili ed eventi	28
3.1.1.2	Funzione createSale	29
3.1.1.3	Funzioni getAllSalesAddresses e getPosessed- SalesAddresses	32
3.1.1.4	Funzione cancelBatchSales	33
3.1.2	Smart Contract TokenSale	33
3.1.2.1	Variabili ed eventi	33
3.1.2.2	Funzione costruttore	35
3.1.2.3	Funzione acceptSale	37
3.1.2.4	Funzione cancelSale	38
3.1.2.5	Funzioni per ottenere informazioni	39
3.2	Test degli smart contract	40
3.2.1	Variabili e funzioni di supporto	41
3.2.1.1	Variabili di supporto	41
3.2.1.2	Funzioni di supporto	42
3.2.2	Test sulla creazione di una vendita	45
3.2.3	Test sul completamento di una vendita	47
3.2.4	Test sulla cancellazione di una vendita	49
4	Implementazione dell'interfaccia grafica	52
4.1	Struttura dell'applicazione web	52
4.2	Azioni preliminari	53
4.2.1	Impostazione di Metamask	53

4.2.2	Impostazione del file di configurazione	54
4.3	File sale.js	54
4.3.1	saleCreate	54
4.3.2	saleGetAll	57
4.3.3	saleAccept	60
4.3.4	saleCancel	62
4.3.5	saleCancelBatch	63
4.4	Pagina "Create Sale"	64
4.4.1	Caricamento della pagina	65
4.4.2	Box di ricerca	66
4.4.3	Step 1: scelta dei token da vendere	68
4.4.3.1	Inserimento di una quantità di una moneta	68
4.4.3.2	Quantità errata	70
4.4.3.3	Avanzamento al passo 2	72
4.4.4	Step 2: scelta dei token da accettare	72
4.4.5	Step 3: riepilogo finale	73
4.4.5.1	Creazione di una vendita	74
4.5	Pagina "Sales List"	76
4.5.1	Caricamento della pagina	77
4.5.2	Pop-up delle vendite scadute	79
4.5.2.1	Calcolo del rimborso	80
4.5.2.2	Accettazione del rimborso	81
4.5.3	Filtro delle vendite	82
4.5.4	Rappresentazione grafica di una vendita	85
4.5.5	Finestra di dialogo di una vendita	86
4.5.5.1	Eliminazione della vendita	87
4.5.5.2	Accettazione della vendita	87

“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.”

1 Introduzione

1.1 CommonsHood

CommonsHood è un'applicazione web basata su smart contract per blockchain Ethereum che ha lo scopo di fornire alla comunità strumenti per l'inclusione finanziaria e per supportare l'economia locale delle comunità di cittadini. Gli utenti, una volta registrati sulla piattaforma, possono creare monete, ossia token crittografici Ethereum basati sullo standard ERC-20, questi possono rappresentare beni di valore o servizi.

Un'altra funzionalità offerta dall'applicazione è la possibilità di creare crowdsales, questi sono usati per ottenere fondi per finanziare progetti o eventi, dando, in cambio, ai finanziatori monete oppure coupons. Un esempio di un'interazione con la piattaforma potrebbe essere: il negozio di attrezzature da sub *"Sotto il Mar"* crea un account su CommonsHood. Crea, inoltre, una moneta chiamata *"Doblone"*, questa può essere spesa in negozio per acquistare le attrezzature. Oltre alla vendita, il negozio noleggia anche le attrezzature, perciò crea dei coupon che possono essere utilizzati per ottenere i prodotti a noleggio. Inoltre, una stanza ha bisogno di ristrutturazioni perciò viene creata una crowdsale per ottenere il finanziamento necessario, in cambio vengono offerti dei coupon del negozio.

1.2 Blockchain

Una blockchain è un registro aperto e distribuito di dati, strutturato come una catena di blocchi contenenti le transazioni. Le transazioni solitamente rappresentano uno scambio di monete, chiamate token. Ogni blockchain ha un token proprio. I dati risiedono su unità computazionali chiamati *node*,

questi, come mostrato nell'immagine 1, sono interconnessi e comunicano tra loro per mantenere i dati di tutti i nodi aggiornati[14]. Un account su una blockchain è costituito da una coppia di chiavi:

- pubblico: è un indirizzo sulla blockchain, i token nella rete sono registrati come appartenenti ad un indirizzo;
- privato: è come una password che l'utente utilizza per accedere ai propri fondi.

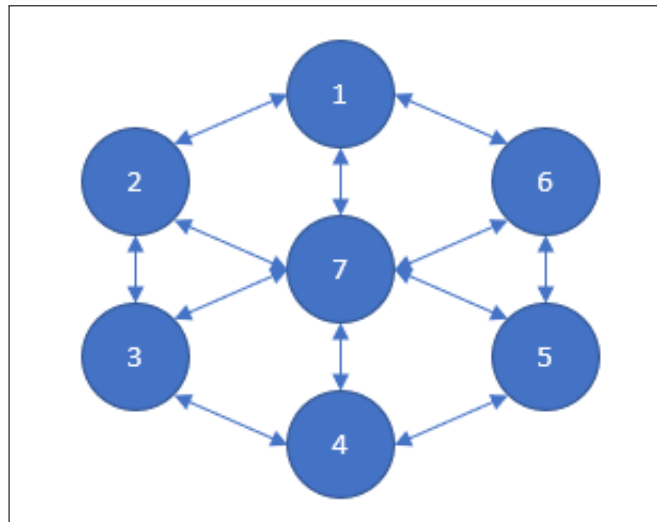


Figura 1: Rete di nodi

1.2.1 I nodi

Le responsabilità di un nodo sono principalmente:

- Controllo della validità di un nuovo record di dati, chiamato anche transazione, e accettarlo o rifiutarlo;
- Nel caso di un record valido, salvataggio della transazione nel registro locale del nodo;

- Comunicazione e distribuzione della transazione agli altri nodi. In questo modo tutti i nodi hanno la stessa versione del registro.

1.2.2 Le caratteristiche di una blockchain

Le caratteristiche principali della tecnologia blockchain sono:

- Decentralizzazione: le informazioni contenute nel registro digitale vengono distribuite tra più nodi, così da garantire sicurezza e resilienza dei sistemi anche in caso di attacco a uno dei nodi o in caso di perdita di un nodo.
- Tracciabilità: ogni elemento salvato nel registro è tracciabile in ogni sua parte e se ne può risalire all'esatta provenienza e alle eventuali modifiche apportate nel corso del tempo, con una precisione assoluta.
- Disintermediazione: i singoli nodi della blockchain certificano le informazioni distribuite, rendendo quindi del tutto inutile la presenza di enti centrali o di aziende per la certificazione dei dati.
- Trasparenza: il contenuto del registro è visibile a tutti ed è facilmente consultabile e verificabile da ogni nodo della rete ma anche tramite servizi che interrogano la blockchain senza apportare modifiche. Nessuno può nascondere o modificare dati senza che l'intera rete venga a saperlo.
- Solidità del registro: dopo aver aggiunto un'informazione al registro, essa non può essere modificata senza il consenso di tutta la rete.
- Programmabilità: le operazioni di transazione possono anche essere programmate nel tempo, così da poter attendere il verificarsi di determinate condizioni prima di procedere con l'inserimento o la modifica[19].

1.2.3 Smart contracts

Le blockchain permettono d'implementare codici e funzioni all'interno di esse, questi sono chiamati smart contracts e permettono l'esecuzione di operazioni quando predeterminate condizioni si avverano. Sono tipicamente usate per automatizzare l'esecuzione di un accordo, in questo modo tutti i partecipanti possono verificarne immediatamente i risultati, senza aver bisogno di un intermediario. Possono, inoltre, essere utilizzate per automatizzare workflow, innescando azioni successive al raggiungimento di certe condizioni[10].

1.3 Ethereum

Ethereum è una piattaforma blockchain decentralizzata che stabilisce una rete peer-to-peer che esegue e verifica smart contracts in modo sicuro. Gli smart contracts permettono transazioni tra gli utenti senza la necessità di un autorità centrale. Le transazioni sono immutabili, verificabili, e distribuiti in modo sicuro sulla rete. Le transazioni sono inviate e ricevute da account Ethereum creati dagli utenti. Come costo per il processamento di una transazione sulla rete, l'utente deve spendere Ether (ETH), la criptovaluta nativa di Ethereum[1].

1.3.1 Token Ethereum

Ethereum permette la creazione di token crittografici all'interno della sua rete. Questi token non sono altro che smart contracts scritti seguendo specifiche stabilite dagli sviluppatori della rete. I token possono essere di tipi differenti, a seconda delle specifiche seguite. In Ethereum ci sono principalmente due tipi di token: token ERC-20 e NFT.

1.3.1.1 Token ERC-20

I token ERC-20 sono il tipo di token più diffuso sulla rete Ethereum, sono lo standard per la definizione di token fungibili[4], ossia i singoli token sono indistinguibili e intercambiabili tra loro. Un token ERC-20 implementa le specifiche indicate nell'EIP-20¹ che richiede nello smart contract del token la presenza di diversi metodi. I più importanti metodi richiesti sono:

- `balanceOf(address _owner)`: restituisce la quantità di token posseduto da `_owner`;
- `transfer(address _to, uint256 _value)`: trasferisce una quantità di token indicata da `_value` all'indirizzo `_to`;
- `transferFrom(address _from, address _to, uint256 _value)`: trasferisce una quantità `_value` di token dall'indirizzo `_from` all'indirizzo `_to`;
- `approve(address _spender, uint256 _value)`: permette all'indirizzo `_spender` di ritirare fino a `_value` token dall'account[20].

1.3.1.2 NFT

I Non Fungible Tokens sono, appunto, token non fungibili, ossia ogni token è univoco e non intercambiabile con un altro. Sono utilizzati per replicare le proprietà tipiche di un oggetto fisico come la scarsità, l'unicità e la possibilità di dimostrare la proprietà del token[9]. Data la natura del token, l'uso più comune di questi token è la creazione di arte digitale[2].

Gli NFT seguono lo standard dettato dall'EIP-721. I metodi più significativi richiesti dall'EIP-721 sono:

¹Ethereum Improvement Proposal

- `balanceOf(address _owner)`: restituisce il numero di NFT posseduti da `_owner`;
- `ownerOf(uint256 _tokenId)`: restituisce il proprietario del NFT identificato da `_tokenId`;
- `safeTransferFrom(address _from,address _to,uint256 _tokenId)`: trasferisce la proprietà del NFT `_tokenId` da `_from` a `_to`. Fallisce se il chiamante della funzione non è il proprietario corrente, un operatore autorizzato o un indirizzato autorizzato per questo NFT. Inoltre, il metodo fallisce se: `_from` non è il proprietario corrente, `_to` è l'indirizzo zero e `_tokenId` non è un NFT valido[3].

1.3.2 Smart contracts in Ethereum

In Ethereum gli smart contracts sono considerati come account, questo significa che hanno un saldo e possono inviare transazioni sulla rete. A differenza di un normale account, però, gli smart contracts non sono controllati dagli utenti ma eseguono il codice con cui sono stati programmati. Gli account user possono interagire con uno smart contract inviando una transazione che effettua una chiamata a una funzione definita nello smart contract. Di default questi contratti non possono essere eliminati e le interazioni con essi sono irreversibili.

Il codice di uno smart contract si scrive utilizzando *Solidity*, un linguaggio object-oriented usato per implementare smart contracts su diverse piattaforme blockchain. Di seguito un esempio di uno smart contract semplice:

```

1      pragma solidity ^0.5.2;
2
3      contract EsempioContratto() {
```

```

4      address public owner;
5
6      event EsempioEvento(
7          uint256 parametroEvento
8      )
9
10     function esempioFunzione(
11         uint256 _parametro
12     ) public returns (uint256) {
13         emit EsempioEvento(_parametro)
14         return _parametro
15     }
16 }

```

Esempio codice di uno smart contract

Solitamente uno smart contract inizia con la dichiarazione della versione di solidity usata per la scrittura del codice, in questo modo il compilatore, se di versione superiore, rifiuta la compilazione del codice. Per far questo si scrive: `pragma solidity [versione]`. Nell'esempio precedente la versione è dichiarata nella prima riga, in questo caso la versione indicata è superiore a 0.5.2.

Una volta dichiarata la versione, inizia il codice dello smart contract, questo viene indicato con `contract [nome contratto] ()`. Prendendo sempre come riferimento l'esempio precedente, alla riga 4 viene dichiarata una variabile chiamata `owner` con visibilità `public` e tipo `address`, ossia un indirizzo Ethereum. Ci sono 4 livelli di visibilità per le variabili e le funzioni:

- **public**: le funzioni possono essere chiamate anche da contratti esterni, per le variabili vengono generate in automatico delle funzioni getter implicite;
- **external**: le funzioni e le variabili possono essere accedute solo dall'e-

sterno e non internamente nel contratto;

- **internal**: le funzioni e le variabili possono essere accedute dentro il contratto stesso e i contratti derivati;
- **private**: visibili solo nel contratto in cui le variabili e le funzioni sono definite[6].

Solidity fornisce numerosi tipi per le variabili, di cui quelle usate in questa tesi sono:

- **bool**: per indicare una variabile booleana;
- **uint**: per indicare un intero senza segno, può avere diverse dimensioni aggiungendo il numero di bits, ad esempio `uint256`;
- **address**: per indicare un indirizzo Ethereum;
- **mapping**: per indicare un dizionario con chiavi di ricerca e valori associati[5].

Alla riga 6 è stato dichiarato un evento, ossia uno strumento utile per fare logging delle transazioni e per permettere agli utenti di mettersi in ascolto di questi eventi. Un evento può contenere dati aggiuntivi, in questo caso l'evento `EsempioEvento` contiene il dato `parametroEvento` di tipo `uint256`. Per emettere un evento si utilizza `emit [nome evento](dati evento)`, come mostrato alla riga 13.

Alla riga 10 è stata dichiarata una funzione di nome `esempioFunzione`, con visibilità `public`, con parametro `_parametro` e che restituisce un valore di tipo `uint256`. Questa funzione esegue solo due operazioni: emette l'evento `EsempioEvento` e restituisce un valore.

1.4 OpenZeppelin

OpenZeppelin è una libreria per lo sviluppo di smart contracts sicuri. Le principali funzionalità fornite da OpenZeppelin sono:

- Implementazione dei diversi standard dei token Ethereum;
- Gestione del controllo degli accessi agli smart contracts;
- Componenti Solidity riutilizzabili per creare smart contracts[13].

1.5 Metamask

Metamask è un'estensione del web browser. Questo software permette di connettere il browser con applicazioni decentralizzate basate sulla piattaforma Ethereum. Metamask permette la gestione di wallet, ossia l'insieme dei token di un utente, Ethereum, la ricezione e l'invio di criptomonete basate su Ethereum, e l'interazione con applicazioni decentralizzate. L'estensione, inoltre, fornisce le API Ethereum web3, in questo modo le applicazioni sono in grado di leggere dati sulla blockchain[11].

1.6 ReactJS

React è una libreria JavaScript open-source per lo sviluppo d'interfacce utente.

1.6.1 Caratteristiche di ReactJS

React fornisce numerosi strumenti che facilitano lo sviluppo di un'interfaccia grafica. Di seguito sono descritti quelli utilizzati in questa tesi.

1.6.1.1 Componenti

I Componenti permettono di suddividere la UI in parti indipendenti, riutilizzabili e di pensare a ognuna di esse in modo isolato. Per definire un componente è necessario implementare una funzione JavaScript, ad esempio:

```
1      function Saluto(props) {  
2          return <h1>Ciao, {props.nome}</h1>;  
3      }
```

Esempio componente

Questo componente accetta un oggetto parametro contenente dati sotto forma di una singola "props", il quale è un oggetto parametro avente dati al suo interno. Per renderizzare un componente bisogna utilizzare la funzione `ReactDOM.render()`, passandole come parametri il componente da visualizzare e il riferimento al componente padre. Se si volesse, quindi, renderizzare il componente `Saluto`, passando "*Martina*" come parametro `nome`, il codice potrebbe essere:

```
1      ReactDOM.render(  
2          <Saluto nome="Martina"/>,  
3          document.getElementById('root')  
4      );
```

Esempio composizione di componenti

I componenti, inoltre, possono essere composte da altri componenti. In questo caso, renderizzando il componente padre, verranno visualizzati anche i componenti figli. Ad esempio, si potrebbe avere un componente `Convenevoli`

che contiene multipli componenti Saluto:

```
1  function Convevoli() {
2    return (
3      <div>
4        <Saluto nome="Sara" />
5        <Saluto nome="Cahal" />
6        <Saluto nome="Edite" />
7      </div>
8    );
9  }
```

Esempio renderizzazione componente

1.6.1.2 Hook state

Un componente React di default è stateless. Usando la funzione `useState()` si può aggiungere uno stato interno a un componente, React preserverà questo stato tra le ri-renderizzazioni. `useState` ritorna una coppia: il valore dello stato corrente e una funzione che ci permette di aggiornarlo. La funzione ha un unico parametro ed è il suo stato iniziale. Ad esempio, se si volesse realizzare un contatore con un bottone che, alla sua pressione, aumenti il valore del contatore, si potrebbe scrivere il seguente codice:

```
1  function Contatore() {
2    const [contatore, setContatore] = useState(0);
3    return (
4      <div>
5        <p>Hai cliccato {contatore} volte</p>
6        <button
7          onClick={() => setContatore(contatore + 1)}>
8          Cliccami
9        </button>
```

```

10         </div>
11     );
12 }

```

Esempio contatore con stato interno

1.6.1.3 Hook effect

Il costrutto `useEffect()` permette l'esecuzione di funzioni a ogni renderizzazione da parte di React. Questa funzione viene utilizzata per effettuare operazioni nei vari stati del ciclo di vita di un componente.

Nel seguente esempio il titolo del documento viene aggiornato all'aumentare del valore del contatore, infatti, a ogni aggiornamento del DOM da parte di React, viene chiamata la funzione passata a `useEffect()`.

```

1  function ContatoreConTitolo() {
2      const [contatore, setContatore] = useState(0);
3
4      useEffect(() => {
5          document.title = `Hai cliccato ${contatore} volte`;
6      });
7
8      return (
9          <div>
10             <p>Hai cliccato {contatore} volte</p>
11             <button
12                 onClick={() => setContatore(contatore + 1)}>
13                 Cliccami
14             </button>
15         </div>
16     );
17 }

```

Esempio uso di `useEffect()`

1.7 Material-UI

Material-UI è una libreria per ReactJS per creare interfacce utente. La libreria contiene al suo interno numerosi componenti grafici, questi sono forniti di un tema di default, per modificare l'aspetto di un componente si può utilizzare la sua proprietà `className`.

Nell'esempio seguente, preso da [12], vengono modificati le dimensione di un componente `<Button>`:

```
1      .Button {  
2          width: "100px",  
3          height: "100px"  
4      }  
5  
6      <Button className="Button">
```

Esempio modifica aspetto di un componente

1.8 Truffle e Ganache

Truffle e Ganache sono entrambi strumenti contenuti all'interno della suite software Truffle. Ganache permette la creazione di una blockchain Ethereum che viene eseguita in locale, semplificando, così, il deploy e il test degli smart contracts. Truffle è un software che facilita lo sviluppo di smart contracts.

I principali comandi di truffle utilizzati sono stati:

- `truffle compile`, per compilare gli smart contracts;
- `truffle test`, per eseguire i file di test;

- `truffle deploy`², per eseguire il deploy degli smart contracts[15].

1.8.1 Creazione di un progetto Truffle

Per creare un progetto con Truffle si utilizza il comando `truffle init` all'interno della sua cartella. Questo genera la struttura del progetto composta da quattro elementi:

- `contracts/`: la cartella che conterrà gli smart contracts;
- `migrations/`: la cartella che conterrà gli scripts per il deploy dei contratti;
- `test/`: la cartella che conterrà i file per eseguire i test sui contratti;
- `truffle-config.js`: il file di configurazione di Truffle.

1.8.2 Compilazione degli smart contracts

Lanciando il comando `truffle compile` nel progetto vengono compilati tutti gli smart contracts, ossia tutti file con estensione `.sol`, presenti all'interno della cartella `contracts/`. Questa operazione genera una nuova directory `build/contracts` contenente un file `.json` per ogni smart contract compilato. Questi file servono per il corretto funzionamento di Truffle quindi la modifica di essi è sconsigliata. Inoltre, contengono le ABI³ degli smart contracts, ossia delle interfacce che stanno tra un programma utente e la blockchain Ethereum. Queste sono necessarie perché gli smart contracts, di cui sono stati fatti il deploy, sono sotto forma di codici binari e i loro dati sarebbero difficilmente comprensibili. Un' ABI, quindi, descrive le funzioni

²oppure `truffle migrate`

³Application Binary Interface

del suo smart contract ed interpreta i dati di questi metodi. Le ABI saranno, perciò, necessarie durante l'implementazione dell'applicazione utente.

1.8.3 Deploy degli smart contracts

Il file `truffle-config.js` permette di definire reti Ethereum che possono essere usate per eseguire il deploy degli smart contracts. Il file ha al suo interno un oggetto `networks`, questo contiene la lista delle reti scritte nel seguente formato:

```
1      <nome_rete>: {  
2          host: <host>,  
3          port: <port>,  
4          network_id: <network_id>  
5      }
```

Esempio di definizione di una rete Ethereum

Dove `host` e `port` indicano l'indirizzo e la porta della rete. `network_id` è, invece, l'identificativo della rete.

Per fare il deploy su una rete specifica si aggiunge l'opzione `--network` al comando `truffle deploy`, perciò, ad esempio, se si volesse fare il deploy degli smart contracts su una rete Ethereum chiamata `main`, il comando completo sarebbe:

```
1      truffle deploy --network main
```

Esempio di deploy specificando la rete

Quest'opzione di specificare la rete di destinazione del deploy è utile perché è possibile effettuare il deploy non sulla rete principale di Ethereum, dove sarebbe presente una tassa di ETH, ma su una rete di test locale.

1.8.4 Test degli smart contracts

Il comando `truffle test` esegue tutti i file di test inclusi nella cartella `test/`. Per eseguire, invece, un solo file di test lo si specifica nel comando. Quindi ad esempio: `truffle test ./percorso/del/test/file.js`. I file di test sono tutti file presenti nella cartella `test/` con una delle seguenti estensioni: `.js`, `.ts`, `.es6`, `.jsx` e `.sol`.

Truffle si avvale di due framework per la scrittura di file di test: *Mocha* e *Chai*[16]. Il primo è un framework per l'esecuzione di test su file Javascript. Il secondo è una libreria per la scrittura di asserzioni, ossia espressioni che indicano i valori attesi alla fine di un test.

Un tipico file di test ha la seguente forma:

```
1      const Contratto = artifacts.require(
2          "<smart_contract_da_testare>");
3      contract("Contratto", async accounts => {
4          describe("<Funzionalità_da_testare>", async () => {
5              it("<Comportamento_da_testare>", async () => {
6                  ....
7                  assert.equal(
8                      <valore_ottenuto>,
9                      <valore_atteso>,
10                     "<messaggio_da_stampare>"
11                 );
12             });
13             it(
14                 ....
15             )
16         })
17     })
```

Esempio di file di test

La riga 1 serve per indicare un file di smart contract usato all'interno del test, per far questo si utilizza `artifacts.require()`. Il metodo `contract()` alla riga 3 serve per indicare il contratto da testare. Il comando ha due funzionalità:

- prima di ogni esecuzione di `contract()`, viene rifatto il deploy dei contratti. In questo modo i diversi file di test vengono eseguiti in modo indipendente tra loro;
- la funzione `contract()` fornisce una lista di account resi disponibili dalla rete Ethereum usata, questi account possono essere usati nei diversi test.

Il metodo `describe()` alla riga 4 serve per indicare una funzionalità del contratto che si vuole testare. La funzione `it()` indica un comportamento della funzionalità che si vuole controllare.

Per effettuare il controllo di un valore ottenuto rispetto ad un valore atteso, si può utilizzare il comando `assert`, questo fornisce numerosi metodi che eseguono controlli di diverso tipo. Nell'esempio si è utilizzato `assert.equal()`, questo controlla che i due parametri passati siano uguali, in caso negativo il test fallisce e viene stampato il messaggio passato come terzo parametro.

2 Progettazione

L'obiettivo della tesi è lo sviluppo di un sistema di scambio di monete all'interno di CommonsHood. Questo sistema è basato interamente sul meccanismo di domanda e offerta. Un utente può, quindi, decidere di non avere più necessità di alcune sue monete, le offre perciò in cambio di altre monete. Un altro utente, che possiede le monete a cui il primo utente è interessato, può accettare lo scambio. A questo punto lo scambio è completato e il primo utente riceve le monete che gli interessavano, il secondo utente riceve le monete offerte dal primo utente. In uno scambio, quindi, sono presenti solo due attori: da un lato c'è l'utente che crea l'offerta dello scambio definendo le monete coinvolte in questo, dall'altro lato c'è un altro utente che è interessato allo scambio e ne accetta i termini. Entrambi, però, hanno un interesse in comune: che lo scambio di monete avvenga correttamente, ovvero i due utenti vogliono assicurarsi che, al termine dello scambio, loro ricevano le monete corrette, ossia vogliono essere certi che l'altro utente non possa imbrogliare e non mandare le monete promesse.

Per fare questo, sono state implementate degli smart contracts a cui sono state affidate la gestione del sistema di scambio. Questi smart contracts definiscono le regole e i criteri di uno smart contracts. Siccome gli smart contracts sono scritti in codice informatico liberamente consultabile, gli utenti sono certi del funzionamento di uno scambio. A questo punto è solo necessario un'interfaccia grafica per consentire agli utenti di usufruire della funzionalità degli scambi.

In questa tesi viene utilizzato il termine "*vendita*" per riferirsi ad uno scambio, le parti di una vendita sono il "*venditore*", colui che crea l'offerta di scambio, e "*compratore*", colui che accetta lo scambio. Le monete coinvolte

in uno scambio sono state chiamate "monete da vendere", ovvero le monete offerte dal venditore, e "monete da accettare", ovvero le monete a cui il venditore è interessato e che il compratore deve possedere per poter accettare lo scambio.

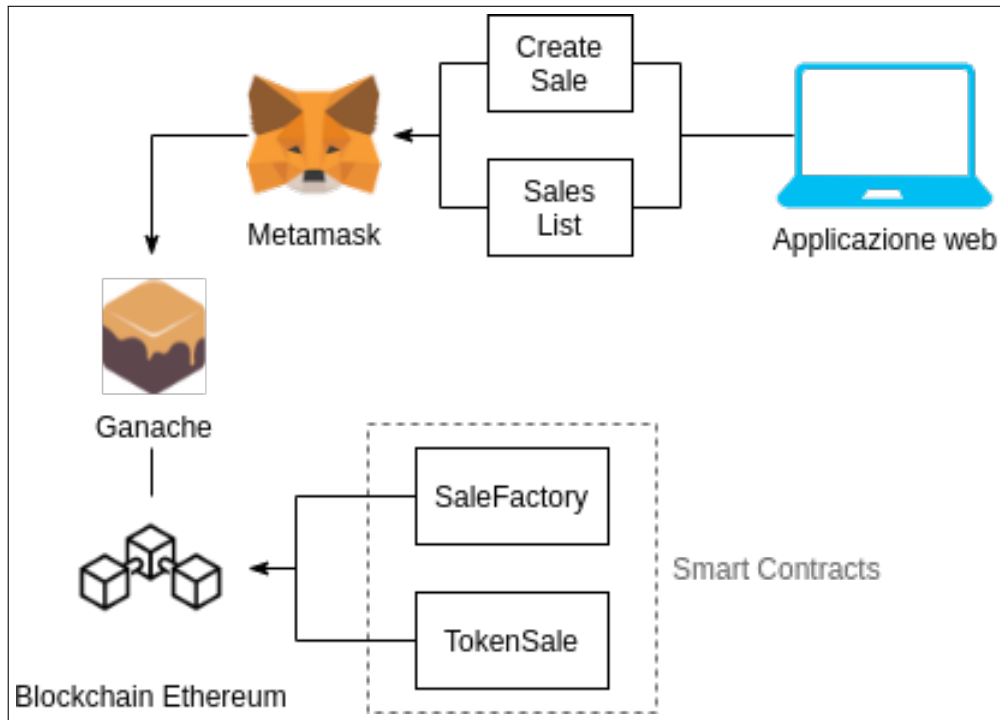


Figura 2: Schema scambio di monete

Nella figura 2 è rappresentato lo schema del funzionamento dello scambio di monete. L'interfaccia web è stata implementata utilizzando *ReactJS* e sfruttando i componenti di *Material-UI*. Sono state create due pagine: "Create Sale" e "Sales List", la prima permette di creare vendite di monete, la seconda permette di visualizzare le vendite e, eventualmente, accettarle o cancellarle. Queste pagine, per svolgere le loro funzioni, hanno bisogno di comunicare con gli smart contracts. Per far ciò utilizzano *Metamask* per connettersi alla blockchain utilizzata. La blockchain è stata creata utilizzando

Ganache e su questa sono stati eseguiti i deploy degli smart contracts che gestiscono la funzionalità dello scambio di monete.

La realizzazione dello scambio di monete è stata divisa in diverse fasi, nella prima sono stati implementati gli smart contracts che regolano una vendita, successivamente si è verificato il corretto funzionamento dei contratti realizzando diversi test, nell'ultima fase è stata creata l'interfaccia utente per interagire con i contratti.

Gli smart contracts sono stati realizzati seguendo il modello "*Factory pattern*", in cui un oggetto "fabbrica" crea oggetti "prodotti". In questo caso il prodotto della fabbrica è la vendita di monete. Sono stati creati, perciò, due smart contracts: **SaleFactory**, ovvero la fabbrica, e **TokenSale**, ovvero lo schema del prodotto. Una vendita rappresenta, quindi, lo scambio di monete tra due utenti: da un lato è presente il venditore, ovvero colui che ha creato la vendita e offre delle monete in cambio di altre monete che sceglie, dall'altro lato è presente il compratore, ovvero colui che accetta i termini della vendita e completa lo scambio. Un utente può interagire in due modi con una vendita: accettarla oppure annullarla. Una vendita si può trovare in tre stati:

- in corso: la vendita è ancora accettabile;
- cancellata: il venditore ha annullato la vendita;
- completata: un utente ha accettato lo scambio.

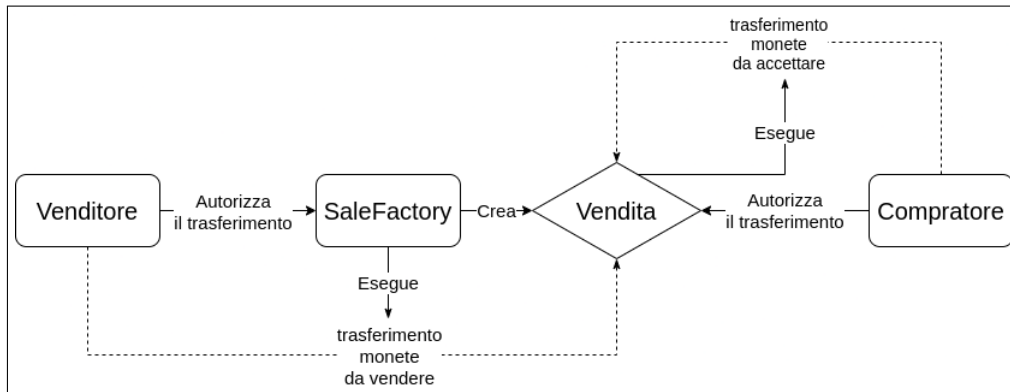


Figura 3: Creazione e accettazione di una vendita

Quando un venditore crea una vendita deve inviare al suo indirizzo le monete che offre. In questo modo l'utente non può spendere le monete che si trovano già in una vendita. Per assicurarsi che il trasferimento delle monete avvenga, questo viene eseguito dal contratto **SaleFactory**, perciò l'utente deve autorizzare la fabbrica a inviare le monete prima che la vendita sia ancora creata. Similarmente, anche quando un utente vuole accettare una vendita deve autorizzare il trasferimento delle monete accettate dalla vendita, ma, siccome l'esecutore del trasferimento è in questo caso la vendita, l'approvazione deve essere data all'indirizzo della vendita. Questo è visibile nello schema in figura 3.

Per evitare errori ed eventuali abusi del sistema da parte di utenti, sono state poste delle restrizioni nella gestione delle vendite. In dettaglio si è impedita la creazione di vendite con quantità di monete negative o nulle, vendite con monete da vendere o accettate duplicate. Viene inoltre impedita la creazione di una vendita se l'utente non ha approvato il trasferimento delle monete da vendere. Un utente, per accettare una vendita, non deve essere il proprietario della vendita, inoltre deve aver autorizzato la vendita a trasferire le monete accettate. Un utente non può accettare una vendita già completata oppure

annullata.

3 Scrittura e test degli smart contracts

In questa fase sono stati, inizialmente, implementati gli smart contracts che realizzano la funzionalità dell'exchange. Successivamente sono stati realizzati i test per verificare il corretto funzionamento dei contratti implementati.

3.1 Scrittura degli smart contracts

Per la scrittura degli smart contracts è stato usato come codice di base i contratti di CommonsHood che implementano la funzionalità della creazione di monete. Gli smart contracts sono stati realizzati seguendo il modello "*Factory Pattern*" per seguire lo standard utilizzato dagli altri contratti di CommonsHood. Quindi, seguendo il pattern, sono stati realizzati due smart contracts: la fabbrica `SaleFactory`, il quale ha il compito di creare le vendite, e il prodotto `TokenSale`, ossia la vendita vera e propria.

3.1.1 Smart Contract `SaleFactory`

Questo contratto ha il compito di creare e tener traccia delle vendite.

3.1.1.1 Variabili ed eventi

```
1 address[] internal saleAddresses;  
2 mapping(address => address[]) salesByOwner;  
3  
4 event SaleAdded(  
5     address indexed _from,  
6     uint256[] amountsOnSale,  
7     uint256[] amountsToAccept,
```

```

8     address owner,
9     address saleAddr
10 );

```

Variabili ed eventi di SaleFactory

Per tener traccia delle vendite create, **SaleFactory** si avvale di due variabili:

- **saleAddresses**: è un array che contiene gli indirizzi di tutte le vendite create;
- **salesByOwner**: è un dizionario con chiave di ricerca l'indirizzo di un utente e con valore associato la lista degli indirizzi delle vendite possedute dall'utente.

Alla creazione di una vendita viene generato un evento **SaleAdded** che ha come dati l'indirizzo del creatore della vendita, le quantità in vendita e le quantità da accettare come pagamento, l'indirizzo del proprietario della vendita e l'indirizzo della vendita.

3.1.1.2 Funzione createSale

```

1 function createSale(
2     address[] memory _tokensOnSale,
3     address[] memory _tokensToAccept,
4     uint256[] memory _amountsOnSale,
5     uint256[] memory _amountsToAccept,
6     uint256 _expiration
7 ) public returns(address) {
8     require(_tokensOnSale.length == _amountsOnSale.length, "All tokens on
        sale must have an amount on sale");
9
10    require(_tokensToAccept.length == _amountsToAccept.length, "All tokens
        to accept must have an amount to accept");

```

```

11
12 for(uint256 i = 0; i < _tokensOnSale.length; i++) {
13     require(_amountsOnSale[i] > 0, "Amounts on sale must be higher than 0"
14 );
15
16     require(TokenTemplate(_tokensOnSale[i]).allowance(msg.sender, address(
17 this)) >= _amountsOnSale[i], "Seller has approved tokens on sale");
18
19     for(uint256 j = i+1; j < _tokensOnSale.length; j++) {
20         require(_tokensOnSale[i] != _tokensOnSale[j], "Tokens on sale must
21 be different");
22     }
23 }
24
25 for(uint256 i = 0; i < _tokensToAccept.length; i++) {
26     require(_amountsToAccept[i] > 0, "Amounts to accept must be higher
27 than 0");
28
29     for(uint256 j = i+1; j < _tokensToAccept.length; j++) {
30         require(_tokensToAccept[i] != _tokensToAccept[j], "Tokens to accept
31 must be different");
32     }
33 }
34
35 for(uint256 i = 0; i < _tokensOnSale.length; i++) {
36     for(uint256 j = 0; j < _tokensToAccept.length; j++) {
37         require(_tokensOnSale[i] != _tokensToAccept[j], "Tokens on sale must
38 be different from tokens to accept");
39     }
40 }
41
42 address saleAddr = address(new TokenSale(
43     _tokensOnSale,
44     _tokensToAccept,
45     msg.sender,
46     _amountsOnSale,
47     _amountsToAccept,
48     _expiration,
49     address(this)
50 ));

```

```

46 for(uint256 i = 0; i < _tokensOnSale.length; i++) {
47     TokenTemplate(_tokensOnSale[i]).transferFrom(msg.sender, saleAddr,
48         _amountsOnSale[i]);
49 }
50 saleAddresses.push(saleAddr);
51
52
53 address[] storage ownedSales = salesByOwner[msg.sender];
54 ownedSales.push(saleAddr);
55 salesByOwner[msg.sender] = ownedSales;
56
57 emit SaleAdded(msg.sender, _amountsOnSale, _amountsToAccept, msg.sender,
58     saleAddr);
59 return saleAddr;
60 }

```

Funzione createSale()

Questa funzione permette la creazione di una vendita. Prende in input i dati della vendita da creare, ossia gli indirizzi delle monete da vendere e da accettare, le loro quantità associate e la data di scadenza della vendita. Prima di creare la vendita vengono effettuati diversi controlli. Viene controllato che la lista delle monete in vendita sia di lunghezza uguale a quella della lista delle quantità delle monete in vendita. Lo stesso controllo viene eseguito poi sulle monete da accettare. Successivamente, viene controllato che tutte le quantità delle monete in vendita siano maggiori di 0, che il venditore abbia approvato la spesa, ossia che **SaleFactory** abbia la **allowance** necessaria, e che le monete in vendita siano diverse tra loro. Gli ultimi tre controlli, tranne quello della **allowance**, vengono eseguiti anche per le monete da accettare. L'ultimo controllo verifica che le monete in vendita siano diverse da quelle da accettare. Superati i controlli viene effettivamente creata la vendita creando una nuova istanza del contratto **TokenSale**, passando al costruttore di

`TokenSale` i parametri di `createSale()`, l'indirizzo del mittente della richiesta di creazione, ossia colui che sarà il proprietario della vendita, e l'indirizzo di `SaleFactory`. Viene salvato, inoltre, nella variabile `saleAddr` l'indirizzo della vendita appena creata. Questa sarà utile per le operazioni successive. Una volta creata la vendita, a questa vengono trasferite le quantità di monete in vendita. In questo modo si impedisce al venditore di utilizzare le monete in vendita mentre è ancora in corso la vendita. L'indirizzo della vendita viene, quindi, memorizzata in `saleAddresses`, mediante la funzione `push()` dell'array, e in `ownedSales`, con chiave l'indirizzo del venditore e come valore l'array già presente ma con l'aggiunta della nuova vendita. Viene infine generato l'evento `SaleAdded`.

3.1.1.3 Funzioni `getAllSalesAddresses` e `getPosesedSalesAddresses`

```
1 function getAllSalesAddresses() public view returns(address[] memory) {  
2     return saleAddresses;  
3 }
```

Funzione `getAllSalesAddresses()`

```
1 function getPosesedSalesAddresses(address _possessor) public view returns  
    (address[] memory) {  
2     return salesByOwner[_possessor];  
3 }
```

Funzione `getPosesedSalesAddresses()`

Entrambe queste funzioni hanno la dicitura **view** in quanto non modificano i dati nella blockchain, ma semplicemente restituiscono delle informazioni. **getAllSalesAddresses** restituisce la lista di tutti gli indirizzi delle vendite. **getPossesedSalesAddresses** prende come parametro un indirizzo e restituisce gli indirizzi delle vendite appartenenti a questo.

3.1.1.4 Funzione **cancelBatchSales**

```
1 function cancelBatchSales(address[] memory _salesToCancel) public {  
2     for(uint256 i = 0; i < _salesToCancel.length; i++) {  
3         require(msg.sender == TokenSale(_salesToCancel[i]).getSaleOwner(), "  
4             User must be the owner of the sale");  
5         TokenSale(_salesToCancel[i]).cancelSale();  
6     }  
}
```

Funzione **cancelBatchSales()**

Questa funzione permette la cancellazione di multiple vendite in una volta sola. Prende come parametro la lista delle vendite da eliminare e, per ognuna di queste, Viene quindi chiamato il metodo di eliminazione su ogni vendita.

3.1.2 Smart Contract **TokenSale**

Il contratto **TokenSale** rappresenta la struttura di una vendita e contiene le funzioni che consentono l'interazione con essa.

3.1.2.1 Variabili ed eventi

```
1 TokenTemplate[] tokensOnSale;  
2 TokenTemplate[] tokensToAccept;
```

```

3
4 address[] public tokensOnSaleAddr;
5 address[] public tokensToAcceptAddr;
6
7 mapping(address => uint256) amountsOnSale;
8 mapping(address => uint256) amountsToAccept;
9
10 address public owner;
11 uint256 public expiration;
12 bool public saleEnded;
13
14 address public factoryAddress;
15
16 event SaleCompleted(
17     address indexed seller,
18     address indexed buyer
19 );
20
21 event SaleCancelled(
22     address indexed cancelledBy,
23     uint256 indexed date
24 );

```

Variabili ed eventi di TokenSale

Le variabili utilizzate da `TokenSale` sono le seguenti:

- `tokensOnSale`: array delle monete in vendita;
- `tokensToAccept`: array delle monete da accettare;
- `tokensOnSaleAddr`: array degli indirizzi delle monete in vendita;
- `tokensToAcceptAddr`: array degli indirizzi delle monete da accettare;
- `amountsOnSale`: mapping che, dato l'indirizzo di una moneta in vendita ne restituisce la quantità in vendita;

- `amountsToAccept`: mapping che, dato l'indirizzo di una moneta da accettare ne restituisce la quantità da accettare;
- `owner`: indirizzo del proprietario della vendita;
- `expiration`: data di scadenza della vendita sotto forma di *unix timestamp*, ovvero il numero di secondi passanti dal 1° Gennaio 1970;
- `saleEnded`: variabile di tipo `boolean` che ha valore `true` se la vendita è terminata, `false` altrimenti;
- `factoryAddress`: indirizzo di `SaleFactory`.

`TokenSale` può generare due tipi eventi:

- `SaleCompleted`: viene creato questo evento quando la vendita è completata, ossia quando un utente accetta la vendita. Le informazioni contenute nell'evento sono l'indirizzo del proprietario della vendita e l'indirizzo del compratore;
- `SaleCancelled`: questo evento viene generato quando la vendita viene eliminata. Le informazioni contenute sono l'indirizzo dell'utente che ha cancellato la vendita e la data di cancellazione.

3.1.2.2 Funzione costruttore

```

1 constructor(
2   address[] memory _tokensOnSale,
3   address[] memory _tokensToAccept,
4   address _owner,
5   uint256[] memory _amountsOnSale,
6   uint256[] memory _amountsToAccept,
7   uint256 _expiration,

```

```

8   address _factoryAddress
9 ) public{
10  for(uint256 i = 0; i < _tokensOnSale.length; i++) {
11      tokensOnSale.push(TokenTemplate(_tokensOnSale[i]));
12      tokensOnSaleAddr.push(_tokensOnSale[i]);
13      amountsOnSale[_tokensOnSale[i]] = _amountsOnSale[i];
14  }
15  for(uint256 i = 0; i < _tokensToAccept.length; i++) {
16      tokensToAccept.push(TokenTemplate(_tokensToAccept[i]));
17      tokensToAcceptAddr.push(_tokensToAccept[i]);
18      amountsToAccept[_tokensToAccept[i]] = _amountsToAccept[i];
19  }
20
21  owner = _owner;
22  expiration = _expiration;
23  factoryAddress = _factoryAddress;
24  saleEnded = false;
25 }

```

Funzione costruttore di TokenSale

Questa funzione rappresenta il costruttore di **TokenSale**, ossia è il metodo eseguito alla creazione di una nuova istanza di **TokenSale**, ovvero alla creazione di una vendita. Prende come parametri la lista degli indirizzi delle monete in vendita e da accettare, le loro quantità, l'indirizzo del proprietario della vendita, la data di scadenza sotto forma di *timestamp* e l'indirizzo di **SaleFactory**. Non è necessario fare i controlli sui parametri in quanto sono già stati verificati nel metodo chiamante **createSale()** contenuto in **SaleFactory**. Per ogni indirizzo delle monete in vendita si ottiene l'istanza della moneta e questa viene inserita nell'array **tokensOnSale**, l'indirizzo viene invece inserito in **tokensOnSaleAddr**, nel mapping **amountsOnSale** viene associato all'indirizzo la quantità in vendita della moneta. Lo stesso viene eseguito anche per gli indirizzi delle monete da accettare. Successivamente ven-

gono assegnati i valori alle variabili `owner`, `expiration` e `factoryAddress` con i parametri passati. In quanto una vendita nuova per definizione non è completata, `saleEnded` è inizializzata a `false`.

3.1.2.3 Funzione `acceptSale`

```
1 function acceptSale() public {
2   require(saleEnded == false, "Sale has to be still on going");
3   require(owner != msg.sender, "Buyer has to be different than the sale
   owner");
4
5   for(uint256 i = 0; i < tokensToAccept.length; i++) {
6     require(tokensToAccept[i].allowance(msg.sender, address(this)) >=
       amountsToAccept[tokensToAcceptAddr[i]], "Buyer has approved the sale
       contract to transfer enough tokens to accept");
7     tokensToAccept[i].transferFrom(msg.sender, owner, amountsToAccept[
       tokensToAcceptAddr[i]]);
8   }
9
10  for(uint256 i = 0; i < tokensOnSale.length; i++) {
11    tokensOnSale[i].transfer(msg.sender, amountsOnSale[tokensOnSaleAddr[i]
      ]]);
12  }
13
14  emit SaleCompleted(owner, msg.sender);
15  saleEnded = true;
16 }
```

Funzione `acceptSale()`

Questa funzione permette al chiamante di accettare la vendita su cui è stata chiamata. La funzione controlla che la vendita non sia già terminata e che il chiamante non sia il venditore stesso. Inoltre verifica che il contratto abbia l'autorizzazione del compratore a spendere le sue monete. Superati i controlli

vengono trasferite le monete interessate dalla vendita. Quindi le monete in vendita vengono trasferite dal contratto all'indirizzo del compratore, mentre le monete da accettare vengono trasferite dal compratore al proprietario della vendita. Infine viene emesso l'evento `SaleCompleted` indicando gli indirizzi di venditore e compratore e, siccome la vendita è conclusa, `saleEnded` viene posto a `true`.

3.1.2.4 Funzione `cancelSale`

```
1 function cancelSale() public {
2     require(msg.sender == owner || msg.sender == factoryAddress, "Only the
3         sale creator or the factory can cancel it");
4     require(saleEnded == false, "Sale has to be still on going");
5
6     for(uint256 i = 0; i < tokensOnSale.length; i++) {
7         tokensOnSale[i].transfer(owner, amountsOnSale[tokensOnSaleAddr[i]]);
8     }
9
10    saleEnded = true;
11
12    emit SaleCancelled(msg.sender, now);
13 }
```

Funzione `cancelSale()`

Questa funzione permette al venditore di annullare la vendita su cui è chiamata. Verifica che il chiamante sia il proprietario della vendita e che la vendita sia ancora in corso. Successivamente restituisce le monete al vendite. Quindi trasferisce i fondi dal contratto al proprietario. Infine pone `saleEnded` a `true` e emette l'evento `SaleCancelled` indicando il chiamante della funzione e la data della chiamata.

3.1.2.5 Funzioni per ottenere informazioni

```
1 function getTokensOnSaleAddr() public view returns(address[] memory) {
2     return tokensOnSaleAddr;
3 }
4
5 function getTokensToAcceptAddr() public view returns(address[] memory) {
6     return tokensToAcceptAddr;
7 }
8
9 function getSaleExpiration() public view returns(uint256) {
10    return expiration;
11 }
12
13 function getSaleOwner() public view returns(address) {
14    return owner;
15 }
16
17 function getSaleInfo() public view returns(address, bool, address[] memory
18     , uint256[] memory, address[] memory, uint256[] memory, uint256) {
19     uint256[] memory retAmountsOnSale = new uint256[](tokensOnSaleAddr.
20         length);
21     uint256[] memory retAmountsToAccept = new uint256[](tokensToAcceptAddr.
22         length);
23
24     for(uint256 i = 0; i < tokensOnSaleAddr.length; i++) {
25         retAmountsOnSale[i] = amountsOnSale[tokensOnSaleAddr[i]];
26     }
27
28     for(uint256 i = 0; i < tokensToAcceptAddr.length; i++) {
29         retAmountsToAccept[i] = amountsToAccept[tokensToAcceptAddr[i]];
30     }
31
32     uint256 exp = expiration;
33
34     return (
35         owner,
36         saleEnded,
37         tokensOnSaleAddr,
```



```

35     retAmountsOnSale,
36     tokensToAcceptAddr,
37     retAmountsToAccept,
38     exp
39 );
40 }

```

Funzioni per ottenere informazioni

Queste funzioni restituiscono informazioni sulla vendita su cui sono chiamate. Siccome non modificano dati hanno tutte la dicitura **view**.

- `getTokensOnSaleAddr()`: restituisce l'array delle monete in vendita;
- `getTokensToAcceptAddr()`: restituisce l'array delle monete da accettare;
- `getSaleExpiration()`: restituisce la data di scadenza della vendita;
- `getSaleOwner()`: restituisce l'indirizzo del venditore;
- `getSaleInfo()`: restituisce tutte le informazioni sulla vendita. Siccome in Solidity le funzioni non possono restituire oggetti di tipo **mapping**, i valori di `amountsOnSale` e `amountsToAccept` sono stati prima trasferiti in due array.

3.2 Test degli smart contract

Per effettuare il test degli smart contracts descritti in 3.1 è stato creato il file `SaleFactoryAndTemplate.js` all'interno della cartella `test/`. In questo sono stati implementati i vari test per verificare il corretto comportamento dei contratti. I test sono stati suddivisi nelle tre funzionalità primarie da

verificare: la creazione di una vendita, il completamento di una vendita e la cancellazione di una vendita.

3.2.1 Variabili e funzioni di supporto

Per facilitare la scrittura dei test sono stati definite alcune variabili e funzioni di supporto.

3.2.1.1 Variabili di supporto

```
1  const tokenOnSaleOne = {
2    name: "Sale Accept Token",
3    symbol: "CRACC",
4    decimals: 18,
5    logoURL:
6      "https://apollo-uploads-las.s3.amazonaws.com/1442324623/atlanta-hawks-
7      logo-944556.png",
8    logoHash: web3.utils.toHex(
9      "0x4D021B157A49F472A48AB02A1F2F6E2986C169A7C78CC94179EDAEBD5E96E8E4"
10   ),
11    contractHash: web3.utils.toHex(
12      "0x4D021B157A49F472A48AB02A1F2F6E2986C169A7C78CC94179EDAEBD5E96E8E4"
13   ),
14    supply: 100000
15  };
16
17  const tokenOnSaleTwo = {
18    name: "Sale Accept Token Two",
19    symbol: "CRACT",
20    ...
21    supply: 100000
22  };
23
24  const tokenToAcceptOne = {
25    name: "Sale Give Token",
26    symbol: "CRGIV",
27    ...
```

```

27   supply: 100000
28 };
29
30 const tokenToAcceptTwo = {
31   name: "Sale Give Token Two",
32   symbol: "CRGIT",
33   ...
34   supply: 100000
35 };
36
37 const amountsOnSale = [1, 1];
38 const amountsToAccept = [1, 1];
39
40 const seller = accounts[0];
41 const buyer = accounts[1];

```

Variabili di supporto nel test

Sono state definite quattro monete da utilizzare per le vendite, due per le monete da vendere e due per le monete accettate. Le quantità delle monete in vendita e accettate sono state impostate a 1 e indicate da `amountsOnSale` e `amountsToAccept`. `seller` e `buyer` contengono due indirizzi forniti dall'ambiente di test, il primo verrà usato per creare le vendite mentre il secondo per accettarle.

3.2.1.2 Funzioni di supporto

```

1  async function setupContracts(seller, buyer, tokenOnSaleOne,
2    tokenOnSaleTwo, tokenToAcceptOne, tokenToAcceptTwo) {
3
4    let tokensOnSaleAddr = [];
5    let tokensToAcceptAddr = [];
6
7    let transaction = await TokenFactoryInstance.createToken(

```

```

8     tokenOnSaleOne.name,
9     tokenOnSaleOne.symbol,
10    tokenOnSaleOne.decimals,
11    tokenOnSaleOne.logoURL,
12    tokenOnSaleOne.logoHash,
13    tokenOnSaleOne.supply,
14    tokenOnSaleOne.contractHash,
15    { from: seller }
16 );
17 tokensOnSaleAddr.push(transaction receipt.logs[0].args._contractAddress)
18 ;
19 transaction = await TokenFactoryInstance.createToken(
20     ...
21     { from: seller }
22 );
23 tokensOnSaleAddr.push(transaction receipt.logs[0].args._contractAddress)
24 ;
25 transaction = await TokenFactoryInstance.createToken(
26     ...
27     { from: buyer }
28 );
29 tokensToAcceptAddr.push(transaction receipt.logs[0].args.
30     _contractAddress);
31 transaction = await TokenFactoryInstance.createToken(
32     ...
33     { from: buyer }
34 );
35 tokensToAcceptAddr.push(transaction receipt.logs[0].args.
36     _contractAddress);
37 return [tokensOnSaleAddr, tokensToAcceptAddr];
38 }
39
40 async function createSale(seller, tokensOnSaleAddr, tokensToAcceptAddr,
41     amountsOnSale, amountsToAccept) {
42     const saleFactoryInstance = await SaleFactory.new();
43     for(const tokenOnSaleAddr of tokensOnSaleAddr) {

```

```

44     let tokenOnSaleInstance = await TokenTemplate.at(tokenOnSaleAddr);
45
46     await tokenOnSaleInstance.approve(saleFactoryInstance.address,
47     amountsOnSale[0], { from: seller });
48 }
49
50 const expiration = 0;
51
52 const transaction = await saleFactoryInstance.createSale(
53     tokensOnSaleAddr,
54     tokensToAcceptAddr,
55     amountsOnSale,
56     amountsToAccept,
57     expiration,
58     { from: seller }
59 )
60 const saleAddr = transaction.receipt.logs[0].args.saleAddr;
61
62 return saleAddr
63 }
64
65 const assertFailError = error => {
66     assert.fail(`
67     Should always be able to call smart contracts, got instead this
68     error:
69     caused by:
70     ${error.stack}
71     Reason: "${error.reason}"`);
72 };

```

Funzioni di supporto nel test

La funzione `setupContracts()` serve per creare le istanze delle monete dichiarate in 3.2.1.1, quelle in vendita vengono create dal venditore mentre quelle da accettare sono create dal compratore. In questo modo entrambi gli account hanno i fondi per eseguire le loro azioni nei test. Per l'istanziatura viene prima creata l'istanza di `TokenFactory`, ovvero lo smart contract re-

sponsabile della creazione di monete, e su questo viene chiamato il metodo `createToken()` passando i dati della moneta da creare. La funzione restituisce gli array degli indirizzi delle monete in vendita e da accettare.

La funzione `createSale()` è usata per creare le vendite. Viene prima creata l'istanza di `SaleFactory`, il contratto responsabile per la creazione di vendite. Successivamente il venditore approva l'istanza di `SaleFactory` a spendere le monete in vendita. Per far questo, sulle istanze delle monete viene chiamata la funzione `approve` e indicando, usando la dicitura `from`, l'indirizzo del venditore come mittente della chiamata. Per comodità la data di scadenza è stata impostata a 0, ossia la vendita non ha data di scadenza. Viene quindi creata la vendita con il metodo `createSale()` fornito da `SaleFactory` e ne viene restituito l'indirizzo.

La funzione `assertFailError()` provoca il fallimento del test e stampa l'errore passato come parametro.

3.2.2 Test sulla creazione di una vendita

I test della funzionalità di creazione di una vendita sono tutte all'interno di `describe("Sale Creation", ...)`. Prima di iniziare i test vengono istanziate le monete interessate dalla vendita chiamando la funzione `setupContracts()` descritta in 3.2.1.2. Il principale comportamento del contratto da verificare è che crei correttamente la vendita.

```
1 it("Creates Sale", async () => {
2   try {
3     saleAddr = await createSale(seller, tokensOnSaleAddr,
4                               tokensToAcceptAddr, amountsOnSale, amountsToAccept);
5     const tokenSaleInstance = await TokenSale.at(saleAddr);
```

```

6   const toCheckTokensOnSaleAddr = await tokenSaleInstance.
   getTokensOnSaleAddr();
7   const toCheckTokensToAcceptAddr = await tokenSaleInstance.
   getTokensToAcceptAddr();
8
9   for(const tokenOnSaleAddr of tokensOnSaleAddr) {
10      let tokenOnSaleInstance = await TokenTemplate.at(tokenOnSaleAddr);
11
12      assert(toCheckTokensOnSaleAddr.includes(tokenOnSaleAddr), "Sale
   should have the correct tokens on sale");
13      assert.equal(await tokenOnSaleInstance.balanceOf(saleAddr),
   amountsOnSale[0], "Sale contract should have received the amount of
   tokens on sale");
14   }
15
16   for(const tokenToAcceptAddr of tokensToAcceptAddr) {
17      assert(toCheckTokensToAcceptAddr.includes(tokenToAcceptAddr), "Sale
   should have the correct tokens to be accepted");
18   }
19 } catch(error) {
20   assertFailError(error);
21 }
22 });

```

Il contratto crea correttamente una vendita

Questo test crea una vendita e verifica che il contratto della vendita abbia ricevuto le corrette quantità di monete da vendere dal venditore. Controlla inoltre che la vendita abbia impostate correttamente le monete da accettare. In caso di errore nel processo di creazione viene eseguito il blocco `catch` che cattura l'errore e richiama `assertFailError()` che provoca il fallimento del test. Gli altri test eseguiti sono:

- `it("Should not allow the creation of a sale with duplicate token on sale")`: controlla che venga correttamente impedita la creazione di una vendita con monete in vendita duplicate;

- `it("Should not allow the creation of a sale with duplicate token to accept")`: effettua lo stesso controllo precedente ma per le monete da accettare;
- `it("Should not allow the creation of a sale with same token on sale and to accept")`: controlla che venga impedita la creazione di una vendita con una stessa moneta sia in vendita che da accettare.

3.2.3 Test sul completamento di una vendita

Questi test, posizionati all'interno di `describe("Sale completed", ...)`, verificano che una vendita venga accettata dal compratore in modo corretto.

```

1  it("Sells the tokens", async () => {
2  try {
3      let initialSellerBalance = [];
4      let initialBuyerBalance = [];
5
6      for(const tokenOnSaleAddr of tokensOnSaleAddr) {
7          let tokenOnSaleInstance = await TokenTemplate.at(tokenOnSaleAddr);
8
9          initialSellerBalance.push(await tokenOnSaleInstance.balanceOf(seller
10         ));
11     }
12     for(const tokenToAcceptAddr of tokensToAcceptAddr) {
13         let tokenToAcceptInstance = await TokenTemplate.at(tokenToAcceptAddr
14         );
15
16         initialBuyerBalance.push(await tokenToAcceptInstance.balanceOf(buyer
17         ));
18     }
19     saleAddr = await createSale(seller, tokensOnSaleAddr, tokensToAcceptAddr
20         , amountsOnSale, amountsToAccept);

```



```

20  const tokenSaleInstance = await TokenSale.at(saleAddr);
21
22  let i = 0;
23  for(const tokenToAcceptAddr of tokensToAcceptAddr) {
24      let tokenToAcceptInstance = await TokenTemplate.at(tokenToAcceptAddr
25      );
26
27      await tokenToAcceptInstance.approve(saleAddr, amountsToAccept[i], {
28      from: buyer});
29      i++;
30  }
31
32  await tokenSaleInstance.acceptSale({ from: buyer});
33  i = 0;
34  for(const tokenOnSaleAddr of tokensOnSaleAddr) {
35      let tokenOnSaleInstance = await TokenTemplate.at(tokenOnSaleAddr);
36
37      assert.equal(await tokenOnSaleInstance.balanceOf(seller, { from:
38      seller}), initialSellerBalance[i]-amountsOnSale[i], "[POST-SALE]
39      Seller has the correct amount of tokens on sale after the sale");
40      assert.equal(await tokenOnSaleInstance.balanceOf(buyer, { from:
41      buyer}), amountsOnSale[i], "[POST-SALE] Buyer has bought the correct
42      number of tokens");
43      i++;
44  }
45
46  i = 0;
47  for(const tokenToAcceptAddr of tokensToAcceptAddr) {
48      let tokenToAcceptInstance = await TokenTemplate.at(tokenToAcceptAddr
49      );
50
51      assert.equal(await tokenToAcceptInstance.balanceOf(buyer, { from:
52      buyer}), initialBuyerBalance[i]-amountsToAccept[i], "[POST-SALE] Buyer
53      has the correct amount of tokens after the sale");
54      assert.equal(await tokenToAcceptInstance.balanceOf(seller, { from:
55      seller}), amountsToAccept[i], "[POST-SALE] Seller has received the
56      payment");
57      i++;
58  }
59
60  assert.equal(await tokenSaleInstance.saleEnded(), true, "[POST-SALE]

```

```
        Sale has ended");
50 } catch(error) {
51     assertFailError(error);
52 }
53 });
```

Il compratore accetta correttamente una vendita

In questo test il venditore crea la vendita, il compratore approva il contratto della vendita a spendere le monete in vendita e, quindi, accetta la vendita. Il test verifica che il venditore e il compratore abbiano ricevuto le corrette quantità di monete e controlla che la vendita sia nello stato "terminato". I test secondari sono:

- `it("Should not allow the seller to accept the sale")`: controlla che il venditore non possa accettare la sua stessa vendita;
- `it("Should not allow the buyer to accept an ended sale")`: controlla che il compratore non possa accettare una vendita già terminata;
- `it("Should not complete the sale if the buyer has not yet approved the sale contract for the transfer of the tokens to accept")`: controlla che il compratore non possa accettare la vendita se prima non ha approvato il trasferimento dei fondi.

3.2.4 Test sulla cancellazione di una vendita

La funzione `describe("Cancel sale", ...)` contiene i test che verificano il corretto funzionamento della cancellazione di una vendita.

```

1 it("Cancels the sale successfully", async () => {
2   try {
3     let initialSellerBalance = [];
4
5     for(const tokenOnSaleAddr of tokensOnSaleAddr) {
6       let tokenOnSaleInstance = await TokenTemplate.at(tokenOnSaleAddr);
7
8       initialSellerBalance.push(await tokenOnSaleInstance.balanceOf(seller
9     ));
10    }
11
12    saleAddr = await createSale(seller, tokensOnSaleAddr,
13    tokensToAcceptAddr, amountsOnSale, amountsToAccept);
14
15    const tokenSaleInstance = await TokenSale.at(saleAddr);
16
17    let i = 0;
18    for(const tokenOnSaleAddr of tokensOnSaleAddr) {
19      let tokenOnSaleInstance = await TokenTemplate.at(tokenOnSaleAddr);
20
21      assert.equal(await tokenOnSaleInstance.balanceOf(seller, { from:
22      seller}), initialSellerBalance[i] - amountsOnSale[i], "[PRE-CANCEL]
23      Seller transferred the tokens on sale");
24      i++;
25    }
26
27    await tokenSaleInstance.cancelSale({ from: seller});
28
29    i = 0;
30    for(const tokenOnSaleAddr of tokensOnSaleAddr) {
31      let tokenOnSaleInstance = await TokenTemplate.at(tokenOnSaleAddr);
32
33      assert.equal(await tokenOnSaleInstance.balanceOf(seller, { from:
34      seller}), initialSellerBalance[i].toNumber(), "[POST-CANCEL] Seller
35      got refunded");
36      i++;
37    }
38
39    assert.equal(await tokenSaleInstance.saleEnded(), true, "[POST-CANCEL]
40    Sale has ended");
41  }
42 }

```

```
35 } catch(error) {  
36     assertFailError(error);  
37 }  
38 });
```

La vendita viene correttamente cancellata

In questo test il venditore crea una vendita e poi la cancella chiamando la funzione `cancelSale()`. Viene controllato che il venditore, dopo la cancellazione della vendita, riceva indietro le monete trasferite durante la creazione della vendita. Si verifica, inoltre, il termine della vendita.

4 Implementazione dell'interfaccia grafica

L'interfaccia utente che implementa le funzionalità di scambio di token è stata divisa in due pagine. La prima è chiamata *Create Sale* e, scegliendo i token da vendere e quelli da accettare, permette la creazione di una vendita. La seconda è chiamata *Sales List* e permette di visualizzare la lista di tutte le vendite, sia quelle in corso che quelle terminate.

Per implementare l'interfaccia si è utilizzato un boilerplate avente solo la funzionalità di creazione di monete[18]. Da questa base, quindi, è stata sviluppata la funzionalità di vendita di monete.

4.1 Struttura dell'applicazione web

```
Commonshood-frontend/  
├─ src/  
│   ├── APIs/  
│   │   └─ coin.js  
│   ├── assets/  
│   ├── components/  
│   ├── config/  
│   │   └─ config.js  
│   └─ theme/  
│       └─ theme.js
```

Questo albero rappresenta la struttura del progetto che implementa l'interfaccia grafica. La cartella `APIs/` contiene i file che implementano i metodi per comunicare con la blockchain. Inizialmente è presente il solo file `coin.js`, ossia il file che contiene le funzioni che gestiscono i token.

`assets/` contiene le immagini utilizzate dall'applicazione.

`components/` contiene tutti i componenti *ReactJS* creati e utilizzati nelle pagine dell'applicazione.

`config/` è una cartella che contiene il file `config.js`. Quest'ultimo è un file di configurazione che contiene i parametri di *Ganache* per comunicare con la blockchain e gli indirizzi e le ABI degli smart contracts utilizzati.

`theme/` contiene il file `theme.js` che definisce lo schema dei colori utilizzati dall'app.

4.2 Azioni preliminari

Prima di iniziare la fase d'implementazione dell'interfaccia grafica, è necessario impostare correttamente *Metamask* e cambiare i parametri del file di configurazione `config.js` della web app presente nella cartella `src/config`.

4.2.1 Impostazione di Metamask

È necessario impostare correttamente *Metamask* per permettere la corretta connessione con la *blockchain* di test utilizzato nella fase precedente. Per far questo si prende l'informazione di `RPC SERVER` da *Ganache*. Successivamente è necessario creare una nuova rete su *Metamask*. Per far ciò, si entra nelle impostazioni delle reti dell'estensione del browser, si seleziona la voce "Aggiungi Reti". Nella pagina successiva, alla voce "Nuovo URL RPC" si inserisce l'indirizzo `RPC SERVER` dato da *Ganache*, invece alla voce "Chain ID" si inserisce il numero 1337, come descritto dalla documentazione di *Truffle*[17]. Terminata questa operazione è possibile connettersi alla nuova rete collegata, ma è ancora necessario importare almeno un account fornito da *Ganache*. Quindi, dalla pagina "I miei account" di *Metamask* si seleziona la voce "Importa account". Alla voce "Incolla la tua chiave privata qui:", si inserisce la chiave privata di un account di *Ganache*. Per trovare questa chiave è necessario, dalla pagina principale di *Ganache*, selezionare il simbolo

di chiave associato a un account, nella finestra che si apre è presente la chiave privata alla voce "private key".

4.2.2 Impostazione del file di configurazione

Per configurare correttamente il file `config.js`, si modificano le variabili `ETH_POA_NETWORK_ID` e `RPC_ENDPOINT` con le informazioni di "Network ID" e "RPC Server" di *Ganache*. Inoltre, all'oggetto `SMART_CONTRACTS` si aggiungono tre nuove voci:

- `SALE_FCTRY_ADDR`: contiene l'indirizzo del contratto di `saleFactory`;
- `SALE_FCTRY_ABI`: contiene l'ABI di `saleFactory`;
- `SALE_TMPLT_ABI`: contiene l'ABI di `saleTemplate`.

L'informazione dell'indirizzo si prende da *Ganache* nella sezione "Contracts". Le ABI, invece, si prendono dall'output del comando `truffle compile`, come descritto in 1.8.2.

Infine, si deve cambiare anche il valore di `TKN_FCTRY_ADDR` con l'indirizzo di `tokenFactory`, preso, sempre, da *Ganache*.

4.3 File `sale.js`

All'interno della directory `src/APIs/` è stato creato un file chiamato `sale.js`. Lo scopo di questo file è quello di contenere tutti i metodi che effettuano le chiamate alle funzioni degli smart contracts relative alla vendita di token. Per far questo il file importa l'oggetto `SMART_CONTRACTS` da `config.js`.

4.3.1 `saleCreate`

```

1 export const saleCreate = async (web3, sellerAddress, coinsOnSaleAddr,
  amountsOnSale, coinsToAcceptAddr, amountsToAccept, expirationDate) =>
  {
2   try {
3     for(let i = 0; i < coinsOnSaleAddr.length; i++) {
4       const coinInstance = new web3.eth.Contract(
5         SMART_CONTRACTS.TKN_TMPLT_ABI,
6         coinsOnSaleAddr[i],
7       );
8
9       await coinInstance.methods.approve(
10        SMART_CONTRACTS.SALE_FCTRY_ADDR,
11        amountsOnSale[i]
12      ).send({from: sellerAddress, gasPrice: "0"});
13    }
14
15    const SaleFactoryInstance = new web3.eth.Contract(
16      SMART_CONTRACTS.SALE_FCTRY_ABI,
17      SMART_CONTRACTS.SALE_FCTRY_ADDR,
18    );
19
20    const creationResponse = await
21      SaleFactoryInstance.methods.createSale(
22        coinsOnSaleAddr,
23        coinsToAcceptAddr,
24        amountsOnSale,
25        amountsToAccept,
26        expirationDate
27      ).send({from: sellerAddress, gasPrice: "0"});
28
29    return creationResponse.events.SaleAdded.returnValues.saleAddr;
30  } catch (error) {
31    for(let i = 0; i < coinsOnSaleAddr.length; i++) {
32      const coinInstance = new web3.eth.Contract(
33        SMART_CONTRACTS.TKN_TMPLT_ABI,
34        coinsOnSaleAddr[i],
35      );
36
37      await coinInstance.methods.approve(
38        SMART_CONTRACTS.SALE_FCTRY_ADDR, 0

```



```

39     ).send({from: sellerAddress, gasPrice: "0"});
40   }
41   return false;
42 }
43 }

```

Funzione saleCreate()

Questo metodo permette la creazione di una vendita. Prende come parametri l'istanza di `web3`, l'indirizzo del venditore, la lista dei token in vendita e le loro quantità, la lista dei token accettati come pagamento e la loro quantità e la data di scadenza della vendita.

Il corpo della funzione è racchiuso all'interno di un blocco `try/catch`, in questo modo, in caso di fallimento delle chiamate alle funzioni degli smart contracts, si può gestire l'errore e comunicarlo al chiamante.

Il blocco `for` da riga 3 a riga 13 serve per ottenere le istanze dei contratti di ogni token in vendita `coinsOnSaleAddr` e, successivamente, chiamare il metodo `approve()` su essi. Alle chiamate del metodo `approve()` vengono passati come parametri l'indirizzo del contratto `SaleFactory` e la quantità associata al token sulla cui istanza si sta effettuando la chiamata. In questo modo si permette all'utente di approvare `SaleFactory` a spendere per ogni moneta in vendita la loro quantità in vendita.

Le chiamate ad `approve()`, inoltre, hanno bisogno alla fine del metodo `send()`, questo perché le funzioni che modificano lo stato del contratto hanno bisogno di inviare una transazione. Questo viene eseguito, appunto, con `send()` [8].

Alla riga 15 si ottiene l'istanza del smart contract `SaleFactory`. Questo viene effettuato creando un nuovo oggetto di tipo `web3.eth.Contract`, passando come parametri l'ABI e l'indirizzo del contratto. Questo funziona,

naturalmente, solo se è già stato fatto il deploy del contratto di cui si sta cercando di ottenere l'istanza.

Ottenuto l'istanza di `SaleFactory`, si può creare la vendita. Questo è realizzato chiamando alla riga 20 il metodo `SaleFactoryInstance.methods.createSale()`, passando come parametri le opzioni della vendita, ossia le monete in vendita e da accettare, le loro quantità e la data di scadenza.

Infine viene restituito al chiamante l'indirizzo della vendita appena creata. Questo è ottenuto dall'evento `SaleAdded`, preso dalla lista degli eventi nella risposta della chiamata a `SaleFactoryInstance.methods.createSale()`.

In caso di fallimento di un'operazione in questo blocco `try`, l'errore verrebbe gestito dal blocco `catch` alla riga 30. Questo ottiene nuovamente le istanze delle varie monete in vendita e, dunque, per ogni moneta effettua il reset della `allowance` di `SaleFactory`. Questo significa che `SaleFactory` non ha più il permesso di spendere le monete in vendita del venditore. Questo viene effettuato semplicemente usando sempre il metodo `approve()` ma, in questo caso, come quantità viene passato 0.

4.3.2 saleGetAll

```
1 export const saleGetAll = async (web3, accountAddress) => {  
2   let salesList = [];  
3  
4   try {  
5     const SaleFactoryInstance = new web3.eth.Contract(  
6       SMART_CONTRACTS.SALE_FCTRY_ABI,  
7       SMART_CONTRACTS.SALE_FCTRY_ADDR,  
8     );  
9  
10    const salesAddresses = await SaleFactoryInstance.methods.  
    getAllSalesAddresses().call({ from: accountAddress, gasPrice: "0"});  
11  }
```

```

12 if(salesAddresses.length !== 0) {
13     const salesInstances = [];
14
15     for(let i = 0; i < salesAddresses.length; i++) {
16         salesInstances.push(new web3.eth.Contract(
17             SMART_CONTRACTS.SALE_TMPLT_ABI,
18             salesAddresses[i],
19         ));
20     }
21
22     for(let i = 0; i < salesInstances.length; i++) {
23         const saleInfo = await salesInstances[i].methods.getSaleInfo().
24         call({ from: accountAddress, gasPrice: "0"});
25
26         let tokensOnSaleData = [];
27         let tokensToAcceptData = [];
28         const saleOwner = saleInfo[0];
29         const saleEnded = saleInfo[1];
30         const saleExpiration = saleInfo[6];
31
32         const tokensOnSaleAddr = saleInfo[2];
33         const amountsOnSale = saleInfo[3];
34         for(let i = 0; i < tokensOnSaleAddr.length; i++) {
35             const tokenData = await coinGetFullData(web3, accountAddress,
36             tokensOnSaleAddr[i]);
37             tokenData['address'] = tokensOnSaleAddr[i];
38             tokenData['amount'] = amountsOnSale[i];
39             tokensOnSaleData.push(tokenData);
40         }
41
42         const tokensToAcceptAddr = saleInfo[4];
43         const amountsToAccept = saleInfo[5];
44         for(let i = 0; i < tokensToAcceptAddr.length; i++) {
45             const tokenData = await coinGetFullData(web3, accountAddress,
46             tokensToAcceptAddr[i]);
47             tokenData['address'] = tokensToAcceptAddr[i];
48             tokenData['amount'] = amountsToAccept[i];
49             tokensToAcceptData.push(tokenData);
50         }

```

```

49     const saleCompleted = await salesInstances[i].getPastEvents("
SaleCompleted");
50
51     const buyer = saleCompleted[0]?.returnValues.buyer;
52
53     const saleData = {
54         address: salesAddresses[i],
55         owner: saleOwner,
56         ended: saleEnded,
57         buyer: buyer,
58         tokensOnSaleData: tokensOnSaleData,
59         amountsOnSale: amountsOnSale,
60         tokensToAcceptData: tokensToAcceptData,
61         amountsToAccept: amountsToAccept,
62         expiration: saleExpiration
63     }
64     salesList.push(saleData);
65 }
66 }
67 return salesList;
68 } catch(error) {
69     return [];
70 }
71 }

```

Funzione saleGetAll

La funzione `saleGetAll()` restituisce al chiamante la lista di tutte le vendite. Prende come parametri l'istanza di `web3`, l'indirizzo del venditore.

Viene creato inizialmente l'array di ritorno `salesList` vuoto. Alla riga 5 viene presa l'istanza di `SaleFactory`, questa viene usata per ottenere la lista degli indirizzi di tutte le vendite usando la funzione `SaleFactoryInstance.methods.getAllSalesAddresses()`. Quest'ultima ha bisogno alla fine del metodo `call()`. Viene usato questo metodo al posto di `send()` perché la funzione `getAllSalesAddresses()` non altera lo stato dello smart contract,

perciò non c'è necessità di inviare una transazione[7].

Se la lista degli indirizzi delle vendite è di lunghezza 0, ovvero è vuota, la funzione termina restituendo al chiamante l'array vuoto. Altrimenti la funzione prosegue con le operazioni successive.

Alla riga 15 vengono ottenute le istanze di tutti gli indirizzi delle vendite, queste vengono memorizzate all'interno di un array chiamato **salesInstances**. Successivamente viene chiamata la funzione **getAllSalesAddresses()** per ogni istanza di vendita, questa restituisce le informazioni della vendita che vengono salvate in alcune variabili. Tra queste informazioni sono presenti anche gli indirizzi dei token in vendita e da accettare, da questi si devono ottenere anche le informazioni sulle monete. Questo viene eseguito nel blocco **for** alla riga 33: su ogni indirizzo delle monete in vendita viene richiamato il metodo **coinGetFullData()** che restituisce i dati della moneta passata come parametro. L'operazione precedente viene ripetuta nel **for** alla riga 42 per le monete da accettare. Alla riga 49 viene chiamata la funzione **getPastEvents()** su un'istanza di vendita passando come parametro "*SaleCompleted*". Questa funzione restituisce l'evento specificato come parametro, se esiste, altrimenti restituisce **null**. Alla riga 51 alla variabile **buyer** viene assegnato l'indirizzo del compratore della vendita, in caso l'evento **saleCompleted** esista, altrimenti ottiene il valore **null**.

Alla riga 53 viene costruito l'oggetto **saleData** contenente tutte le informazioni sulla vendita e sulle monete. Infine **saleData** viene inserito nell'array **salesList** e quest'ultimo viene restituito al chiamante.

In caso di errore nel blocco **try**, la funzione restituisce una lista vuota.

4.3.3 saleAccept

```

1 export const saleAccept = async (web3, accountAddress, saleAddress,
  coinsToAccept) => {
2   try {
3     for(let coin of coinsToAccept) {
4       const {address: coinAddress, symbol, amount} = coin;
5
6       const tokenInstance = new web3.eth.Contract(
7         SMART_CONTRACTS.TKN_TMPLT_ABI,
8         coinAddress,
9       );
10
11       await tokenInstance.methods.approve(saleAddress, amount).send({from:
        accountAddress, gasPrice: '0'});
12     }
13
14     const saleInstance = new web3.eth.Contract(
15       SMART_CONTRACTS.SALE_TMPLT_ABI,
16       saleAddress,
17     );
18     const res = await saleInstance.methods.acceptSale().send({from:
        accountAddress, gasPrice: '0'});
19     return true;
20   } catch(error) {
21     for(let i = 0; i < coinsToAccept.length; i++) {
22       const coinInstance = new web3.eth.Contract(
23         SMART_CONTRACTS.TKN_TMPLT_ABI,
24         coinsToAccept[i],
25       );
26
27       await coinInstance.methods.approve(saleAddress, 0).send({from:
        accountAddress, gasPrice: "0"});
28     }
29     return false;
30   }
31 }

```

Funzione saleAccept

La funzione `saleAccept` permette di accettare una vendita. Prende come

parametri l'istanza di `web3`, l'indirizzo del compratore, l'indirizzo della vendita e la lista delle monete accettate come pagamento per la vendita.

Per ogni moneta della lista passata come parametro vengono estratte le informazioni di indirizzo della moneta, il simbolo della moneta e la quantità della moneta necessaria per accettare la vendita. Viene, poi, ottenuta l'istanza del token e viene chiamata la funzione `approve` sull'istanza, passando come parametri l'indirizzo della vendita e la quantità della moneta. In questo modo il contratto della vendita ottiene il permesso di spendere la quantità di quella moneta.

Successivamente si ottiene l'istanza della vendita e su questa viene chiamato il metodo `acceptSale()` che conclude l'accettazione della vendita. Viene, quindi, restituito il valore `true` al chiamante. In caso di fallimento viene restituito, invece, `false` e viene fatto il reset della *allowance* del contratto della vendita a 0 per tutte le monete passate in input.

4.3.4 `saleCancel`

```
1 export const saleCancel = async (web3, accountAddress, saleAddress) => {
2   try {
3     const saleInstance = new web3.eth.Contract(
4       SMART_CONTRACTS.SALE_TMPLT_ABI,
5       saleAddress,
6     );
7
8     const res = await saleInstance.methods.cancelSale().send({from:
9       accountAddress, gasPrice: '0'});
10    return true;
11  } catch(error) {
12    return false;
13  }
```

Funzione saleCancel

La funzione `saleCancel` permette di cancellare una vendita. Prende in input l'istanza di `web3`, l'indirizzo del proprietario della vendita e l'indirizzo della vendita da cancellare. Inizialmente viene ottenuta l'istanza della vendita e, successivamente, su di essa viene richiamata la funzione `cancelSale()`. In caso di successo viene restituito `true` altrimenti `false`.

4.3.5 saleCancelBatch

```
1 export const saleCancelBatch = async (web3, accountAddress, saleAddresses)
  => {
2   try {
3     const SaleFactoryInstance = new web3.eth.Contract(
4       SMART_CONTRACTS.SALE_FCTRY_ABI,
5       SMART_CONTRACTS.SALE_FCTRY_ADDR,
6     );
7
8     const res = await SaleFactoryInstance.methods.cancelBatchSales(
9       saleAddresses).send({from: accountAddress, gasPrice: '0'});
10    return true;
11  } catch(error) {
12    return false;
13  }
```

Funzione saleCancelBatch

La funzione `saleCancel` permette di cancellare multiple vendite. Prende in input l'istanza di `web3`, l'indirizzo del proprietario delle vendite e la lista degli indirizzi delle vendite da cancellare.

Viene ottenuta l'istanza di `saleFactory` e su questa viene richiamato il metodo `cancelBatchSales` passando la lista delle vendite. In caso di successo viene restituito `true`, altrimenti `false`.

4.4 Pagina "Create Sale"

Il processo di creazione di una vendita è diviso in tre step.

- Nel primo step l'utente sceglie i token in suo possesso da mettere in vendita;
- Nel secondo step l'utente sceglie i token che accetta come pagamento per la vendita dei token scelti nel primo step;
- Nel terzo step viene mostrato all'utente un riassunto delle scelte fatte nei due step precedenti. In questo passo, inoltre, l'utente può impostare una data di scadenza della vendita.

In tutti e tre gli step sono presenti alcuni componenti grafici comuni, questi sono visibili nell'immagine 4.

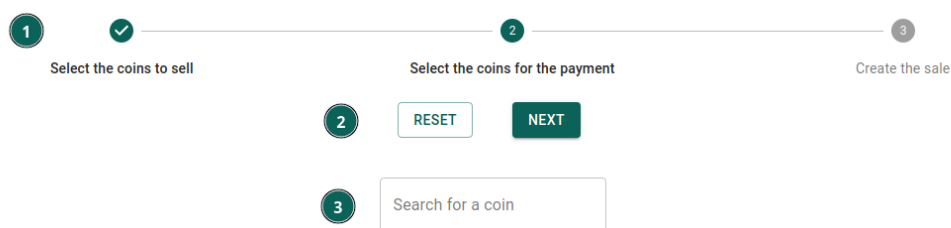


Figura 4: Componenti comuni

Per mostrare graficamente il progresso nei diversi step è presente, in cima alla pagina, un componente `<stepper>`, questo, come mostrato nell'immagine 4 al punto 1, indica all'utente gli step terminati e quelli ancora da completare.

Al di sotto, al punto 2, sono presenti due pulsanti per la navigazione nei vari step. Il primo permette di tornare allo step 1 azzerando tutti i parametri inseriti. Il secondo serve per avanzare allo step successivo.

Al punto 3 è presente un campo di inserimento per cercare una moneta in particolare. Questo componente è presente solo nel primo e secondo step

4.4.1 Caricamento della pagina

Al caricamento della pagina vengono inizializzate alcune variabili, tra cui le più significative sono le seguenti:

```
1      const {web3Instance, userAccount} = props;
2      const [coinsToSellAmounts,
3            setCoinsToSellAmounts] = useState(new Map());
4      const [coinsToAcceptAmounts,
5            setCoinsToAcceptAmounts] = useState(new Map());
6      const [coinsList, setCoinsList] = useState([]);
7      const [activeCoinsList,
8            setActiveCoinsList] = useState([]);
9      const [activeStep, setActiveStep] = useState(0);
```

Inizializzazione delle variabili al caricamento della pagina Create Sale

`web3Instance` contiene l'istanza di web3 mentre `userAccount` contiene l'indirizzo Ethereum dell'utente corrente, questi vengono passati alla pagina come props.

`coinsToSellAmounts` e `coinsToAcceptAmounts` sono variabili dotate di stato ed entrambi sono di tipo `Map`, ossia un dizionario con chiavi di ricerca e valori associati. Vengono usate per mantenere le quantità delle monete scelte da mettere in vendita, per la prima variabile, oppure quelle da accettare come pagamento, nel caso della seconda variabile. Hanno come chiavi di ricerca le

monete mentre i valori sono le quantità di queste.

`coinList` è un array usato per contenere la lista di tutte le monete possedute dall'account utente.

`activeCoinsList` è l'array preso come riferimento per mostrare graficamente la lista delle monete.

`activeStep` è un numero intero che indica lo step corrente.

Le variabili di stato hanno, naturalmente, associate le funzioni per modificarne il loro valore.

Una volta caricata la pagina, con l'uso di `useEffect()`, viene richiamata la funzione `fetchCoins`.

```
1      const fetchCoins = async () => {  
2          setLoadingCoinList(true)  
3          const newCoinsList = await coinGetListOnlyOwned(  
4              web3Instance,userAccount);  
5          setLoadingCoinList(false);  
6          setCoinsList(newCoinsList);  
7          setActiveCoinsList(newCoinsList);  
8      }
```

Funzione `fetchCoins`

4.4.2 Box di ricerca

```
1      const handleSearch = (event) => {  
2          const searchInput = event.target.value.toLowerCase();  
3  
4          setActiveCoinsList(coinsList.filter(coin => {  
5              const coinName = coin.name.toLowerCase();  
6              const coinSymbol = coin.symbol.toLowerCase();  
7              if(coinName.includes(searchInput) ||
```

```

8         coinSymbol.includes(searchInput)) return coin;
9     });
10 }
11
12 <TextField
13   id="searchBox"
14   label="Search for a coin"
15   variant="outlined"
16   onChange={handleSearch}/>

```

Funzione fetchCoins

Il box di ricerca è realizzato usando il componente `<TextField>`, ad ogni inserimento/eliminazione di caratteri viene invocata la funzione `handleSearch`. Questa funzione, come visibile alla riga 1, prende come parametro l'evento lanciato. Quest'ultimo contiene il testo presente nel `<TextField>`, questo viene, quindi, convertito in caratteri minuscoli ed assegnato alla variabile `searchInput`. Alla riga 4 viene chiamata la funzione `filter` sull'array `coinsList`, questo metodo ritorna un nuovo array dopo aver filtrato gli elementi della lista secondo un certo criterio. In questo caso, il criterio, visibile alle righe 7 e 8, è che il nome o il simbolo della moneta che si sta controllando contenga la stringa da cercare contenuta in `searchInput`. In caso positivo la moneta viene aggiunta all'array da ritornare. Una volta terminata la funzione di filtro si ha quindi una lista contenete solo monete che includono il termine di ricerca. Dopodichè questa lista viene assegnata alla variabile di stato `activeCoinsList` usando, perciò, la funzione `setActiveCoinsList()`. In questo modo verranno mostrate a video le monete filtrate.

4.4.3 Step 1: scelta dei token da vendere

1 Select the coins to sell 2 Select the coins for the payment 3 Create the sale

RESET NEXT

Search for a coin

prova - PRO
Balance: 60

mare - MAR
Balance: 100

Amount

Amount

Figura 5: Pagina per la scelta dei token in vendita

La pagina, visibile in figura 5, contiene un componente custom chiamato `SelectCoins`, questo ha come props l'array `activeCoinsList` e genera, usando il componente `<List>`, una lista contenente tutte le monete della lista.

Ogni elemento della lista contiene le informazioni di un token all'interno di `activeCoinsList` e un componente `<TokenListItem>` contenente un componente `<TextField>`. Quest'ultimo permette l'inserimento della quantità desiderata del token da mettere in vendita.

4.4.3.1 Inserimento di una quantità di una moneta

```
1 const handleAmountChange = (coin, amount) => {  
2   const isEmpty = amount.length === 0;  
3   amount = parseInt(amount);  
4 }
```

```

5  if(isInputEmpty || amount === 0) {
6      const newMap = new Map(coinsToSellAmounts);
7      newMap.delete(coin);
8      setCoinsToSellAmounts(newMap);
9  } else {
10     setCoinsToSellAmounts(prev => new Map(prev.set(coin, amount)));
11 }
12
13 if(amount > coin.balance) {
14     setErrorsAmounts(prev => new Set(prev.add(coin)));
15 } else {
16     const newSet = new Set(errorsAmounts);
17     newSet.delete(coin);
18     setErrorsAmounts(newSet);
19 }
20 }

```

Funzione handleAmountChange

All'inserimento della quantità di una moneta viene richiamata la funzione `handleAmountChange(coin, amount)` passando, come parametri, la moneta interessata e la quantità inserita.

La funzione inizia assegnando alla variabile `isInputEmpty` il valore `true` in caso il parametro `amount` sia una stringa vuota, `false` altrimenti. Inoltre, siccome `amount` è una stringa, converte questo in un valore intero.

Successivamente, in caso `isInputEmpty` sia `true` oppure `amount` sia 0, viene eliminato da `coinsToSellAmounts` la voce con chiave di ricerca `coin`. Altrimenti inserisce, sempre in `coinsToSellAmounts`, una voce con chiave `coin` e valore `amount`. In questo modo `coinsToSellAmounts` contiene sempre le monete e le quantità associate inserite dall'utente.

La funzione, inoltre, controlla che la quantità inserita non ecceda la quantità posseduta dall'utente, ossia confronta `amount` e il parametro `balance` all'interno di `coin`.

Nel caso di `amount` maggiore di `coin.balance`, cioè l'utente non possiede la quantità inserita, viene aggiunto a `errorsAmounts` il valore l'oggetto `coin`. Nel caso opposto viene eliminato `coin` da `errorsAmounts`. `errorsAmounts` viene poi utilizzato, in caso di errore nell'inserimento di una quantità, per disabilitare il bottone per passare al prossimo step. Per far questo basta controllare che `errorsAmounts` non sia vuoto.

4.4.3.2 Quantità errata

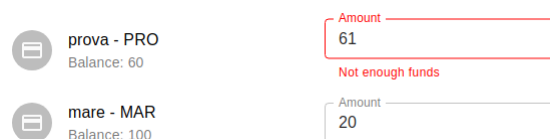


Figura 6: Sopra un inserimento di quantità errata, sotto un inserimento corretto

Il componente `SelectCoins` ha una props chiamata `showCoinError` a cui viene passato il riferimento ad una funzione chiamata `showCoinError()`.

```
1 const showCoinError = (coin, balance) => {  
2   if(activeStep === 0)  
3     if(coinsToSellAmounts.has(coin) && coinsToSellAmounts.get(coin) >  
       balance)  
4       return true;  
5   return false;  
6 }
```

Funzione `showCoinError`

Questa funzione prende in input due parametri: `coin`, ossia una moneta, e `balance`, la quantità di `coin` che l'utente possiede. La funzione inizia controllando che l'attuale step sia lo 0, questo perché è solo nello step iniziale che

possono presentarsi errori nell’inserimento di quantità. La funzione prosegue, quindi, controllando che `coinsToSellAmounts` abbia l’oggetto `coin`, e che il valore associato a `coin` in `coinsToSellAmounts` sia superiore a `balance`. In questo caso vuol dire che l’utente ha inserito `coin` tra le monete da vendere, ma ha inserito una quantità che non possiede, quindi la funzione restituisce `true`.

Il componente `<SelectCoins>` quando genera i vari `<TokenListItem>` di ogni moneta, ossia i componenti che contengono i box di inserimento delle quantità, aggiunge loro una props `showError` contenente la chiamata alla funzione `showCoinError()`, passando la moneta e il bilancio della moneta.

```
1 <TokenListItem
2   showError={props.showCoinError(coin, coin.balance)}
3   ...
4 />
```

Funzione `showCoinError`

Dove `coin` è ottenuto facendo ciclo sull’array `activeCoinsList`.

Ogni `<TokenListItem>` ha un valore di tipo boolean all’interno della proprietà `showError`. Questo viene usato per scegliere se mostrare la `<TextField>` di inserimento con messaggio di errore oppure normale, come visibile in figura 6.

```
1 (props.showError) ?
2 <TextField error={true} helperText={errorMessage} .../>
3 :
4 <TextField .../>
```

Scelta di `TextField` da mostrare

4.4.3.3 Avanzamento al passo 2

Cliccando sul pulsante `next`, oltre a passare al passo successivo, vengono effettuate le seguenti operazioni.

```
1 let newCoinsList = await coinGetListAll(web3Instance, userAccount);
2 const selectedCoins = Array.from(coinsToSellAmounts.keys());
3 newCoinsList = newCoinsList.filter(coin => {
4     for(let c of selectedCoins) {
5         if(c.address === coin.address) return false;
6     }
7     return true;
8 });
9 setLoadingCoinList(false);
10 setCoinsList(newCoinsList);
11 setActiveCoinsList(newCoinsList);
```

Codice eseguito passando al passo 2

Questo codice genera un array `newCoinsList` contenente tutte le monete tranne quelle che l'utente ha selezionato di vendere al passo 1. Questo array viene, quindi, assegnato a `coinsList` e `activeCoinsList`.

4.4.4 Step 2: scelta dei token da accettare

Il processo di scelta dei token da accettare come pagamento è molto simile a quello della scelta dei token da vendere del primo step. La principale differenza è l'array `activeCoinsList` passato a `SelectCoins`. Questo, infatti, nel passo 2 non contiene solo le monete possedute dall'utente, ma contiene tutte le monete tranne quelle che l'utente ha selezionato nello step 1.

Inoltre non è più necessario controllare la quantità inserita perché non c'è la restrizione del bilancio dell'utente, perciò il sistema descritto in 4.4.3.2 non è

applicato in questo step. Naturalmente la funzione `handleAmountChange()`, richiamata ad ogni inserimento/modifica di una quantità, non modifica più `coinsToSellAmounts`, come in 4.4.3.1, ma modifica `coinsToAcceptAmounts`.

4.4.5 Step 3: riepilogo finale

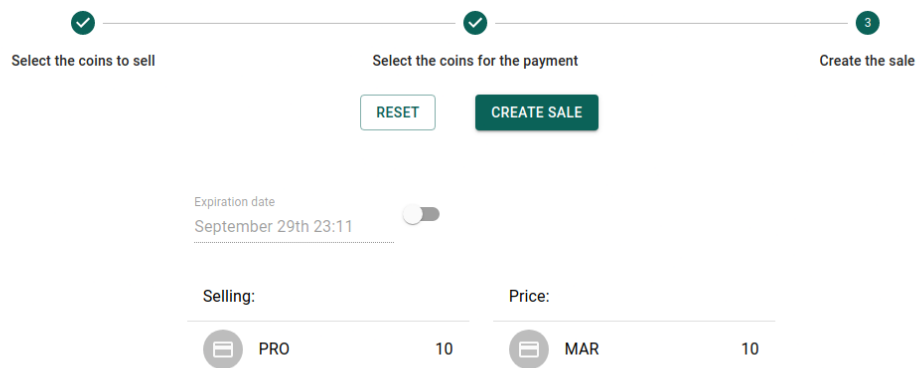


Figura 7: Pagina di riepilogo della creazione di una vendita

In questo step viene mostrato un riepilogo delle scelte fatte dall'utente nei due step precedenti. Questo sommario è implementato da un componente custom chiamato `<ResumePage>` che contiene due componenti `<List>`. I dati delle due liste sono passate a `<ResumePage>` attraverso due sue props: `coinsToSell` e `coinsToAccept`. Al primo viene passato `coinsToSellAmounts`, al secondo viene passato `coinsToAcceptAmounts`. Siccome questi due sono `Map`, prima di utilizzare i loro valori, `<ResumePage>` deve convertirli in `Array`, per fare ciò utilizza la funzione `Array.from()` passando una `Map`. Questa funzione restituisce una lista contenente tutte le voci, quindi le coppie chiave-valore, della `Map` passata.

Il componente `<ResumePage>` contiene, inoltre, un selettore di data, imple-

mentato con il componente `<DateTimePicker>` fornito da *MaterialUI*, e un pulsante per abilitarne o disabilitarne l'inserimento. La data inserita rappresenta la data di scadenza della vendita in corso di creazione, ovvero la data oltre la quale un utente non potrà più accettare la vendita. L'inserimento di una data modifica il valore della variabile di stato `selectedExpirationDate` con il valore della data selezionata sotto forma di "unix timestamp", ovvero la data viene rappresentata come i numeri di secondi passati dal 1° Gennaio 1970 (UTC).

4.4.5.1 Creazione di una vendita

Premendo il pulsante "Create Sale" viene richiamata la funzione `createSale()`.

```
1 const createSale = async () => {
2   const coinsToSellAddr = [];
3   const coinsToSell = Array.from(coinsToSellAmounts.keys());
4   for(let coin of coinsToSell) coinsToSellAddr.push(coin.address);
5
6   const amountsToSell= Array.from(coinsToSellAmounts.values());
7
8   const coinsToAcceptAddr = [];
9   const coinsToAccept= Array.from(coinsToAcceptAmounts.keys());
10  for(let coin of coinsToAccept) coinsToAcceptAddr.push(coin.address);
11
12  const amountsToAccept = Array.from(coinsToAcceptAmounts.values());
13
14  const saleAddr = await saleCreate(web3Instance, userAccount,
    coinsToSellAddr, amountsToSell, coinsToAcceptAddr, amountsToAccept,
    selectedExpirationDate);
15
16  let creationMessage = saleAddr ? "Sale created successfully" : "Sale was
    not created. Something went wrong...";
17
18  handleReset();
19 }
```

```

20  setAlertState({
21      result: saleAddr,
22      creationMessage,
23      open: true,
24  });
25  }

```

Scelta di TextField da mostrare

La funzione, come visibile dalla riga 2 alla riga 6, costruisce due array a partire dal Map `coinsToSellAmounts`: `coinsToSellAddr` e `amountsToSell`. Il primo contiene gli indirizzi delle monete in vendita, il secondo contiene le quantità delle monete in vendita. Questa operazione viene ripetuta anche per `coinsToAcceptAmounts`. Una volta creati gli array, viene richiamata la funzione `saleCreate` contenuta nel file `sale.js`. Vengono passati come parametri l'istanza di `web3`, l'indirizzo dell'utente, l'array degli indirizzi delle monete in vendita, l'array delle quantità in vendita, l'array degli indirizzi delle monete da accettare, l'array delle quantità da accettare e la data di scadenza. In base al risultato della chiamata viene, quindi, mostrato un messaggio di risposta. Infine viene azzerata la pagina, tornando, perciò, allo step 1.

4.5 Pagina "Sales List"

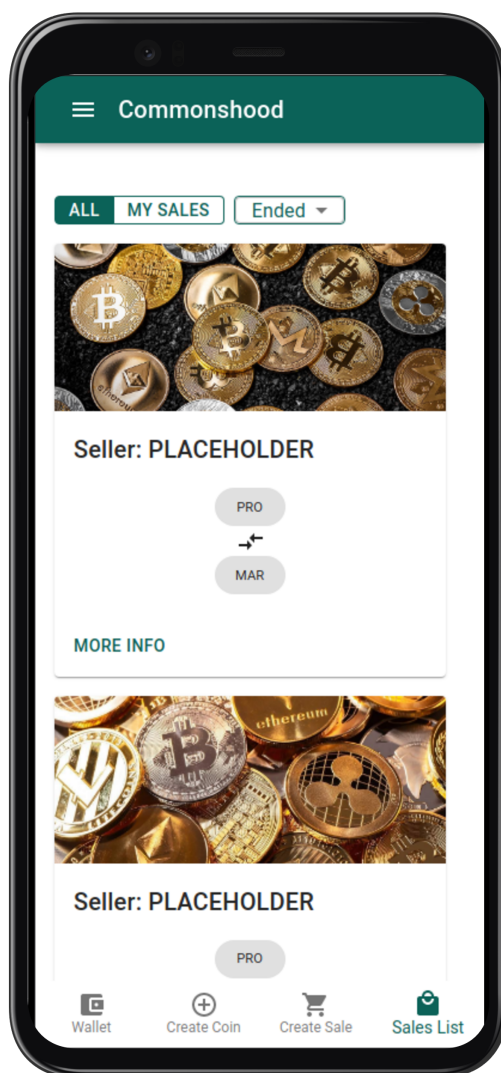


Figura 8: Pagina con la lista delle vendite

Questa pagina permette di visualizzare la lista di tutte le vendite. Inoltre, consente, utilizzando i pulsanti e il selettore in cima alla pagina, di filtrare le vendite secondo diversi parametri. Prima di mostrare le vendite, in caso

l'utente possieda delle vendite scadute, la pagina mostra un *pop-up* che elenca le monete e le loro quantità che l'utente può ottenere indietro.

4.5.1 Caricamento della pagina

Le variabili più significative inizializzate al caricamento della pagina sono le seguenti.

```
1 const [hasExpiredSales, setHasExpiredSales] = useState(false);
2 const [onGoingSalesList, setOnGoingSalesList] = useState([]);
3 const [endedSalesList, setEndedSalesList] = useState([]);
4 const [expiredSalesList, setExpiredSalesList] = useState([]);
5 const [showingSalesList, setShowingSalesList] = useState([]);
6 const [selectedFilter, setSelectedFilter] = useState(0);
7 const [isEndedStatusSelected, setIsEndedStatusSelected] = useState(false);
8 const [selectedSale, setSelectedSale] = useState();
```

Variabili significative di "Sales List"

Tutte queste variabili sono variabili con stato. `hasExpiredSales` è un boolean che indica la presenza di vendite scadute possedute dall'attuale utente. Le loro funzioni sono le seguenti.

`onGoingSalesList` è un array che contiene le vendite ancora in corso, mentre `endedSalesList` contiene tutte le vendite terminate, sia quelle accettate che quelle cancellate, e quelle scadute. Invece, `expiredSalesList` contiene solo le vendite scadute. `showingSalesList` è l'array contenente le vendite visualizzate a video. `selectedFilter` indica il filtro attuale selezionato, mentre `isEndedStatusSelected` indica se si è selezionato di mostrare solo le vendite terminate. `selectedSale` contiene la vendita di cui mostrare le informazioni dettagliate. Al caricamento della pagina viene chiamata la funzione `useEffect()` che, a sua volta, richiama la funzione `fetchSales()`.

```

1  const fetchSales = async () => {
2    setLoadingSaleList(true);
3    const onGoingSales = [];
4    const endedSales = [];
5    const expiredSales = [];
6    let ownerHasExpiredSales = false;
7    let allSalesList = await saleGetAll(web3Instance, userAccount);
8
9    for(let sale of allSalesList) {
10     const isExpired = !sale.ended && sale.expiration !== '0' ? isAfter(new
        Date(), fromUnixTime(sale.expiration)) : false;
11
12     sale.ended || isExpired ? endedSales.push(sale) : onGoingSales.push(
        sale);
13
14     const isExpiredAndOwned = sale.owner === userAccount && isExpired;
15
16     isExpiredAndOwned && expiredSales.push(sale);
17
18     ownerHasExpiredSales = isExpiredAndOwned || ownerHasExpiredSales;
19   }
20
21   setOnGoingSalesList(onGoingSales);
22   setEndedSalesList(endedSales);
23   setExpiredSalesList(expiredSales);
24   setHasExpiredSales(ownerHasExpiredSales);
25   setShowingSalesList(onGoingSales);
26   setLoadingSaleList(false);
27 }

```

Funzione fetchSales()

Questa funzione genera inizialmente una variabile `ownerHasExpiredSales` di tipo booleano inizializzata a `false`, e tre array: `onGoingSales`, `endedSales` e `expiredSales`. Viene assegnata, chiamando la funzione `saleGetAll()`, alla variabile `allSalesList` la lista di tutte le vendite. Per ogni vendita si controlla se è scaduta. Questa verifica, visibile alla riga 10, avviene controllando

che la vendita non sia terminata e che la data di scadenza sia diversa da 0 e che l'attuale data sia successiva alla data di scadenza. Nel caso di vendita scaduta oppure terminata, questa viene inserita in `endedSales`, altrimenti in `onGoingSales`. Successivamente la funzione controlla se la vendita è scaduta ed è posseduta dall'utente, in questo caso la vendita viene inserita anche in `expiredSales`. Viene poi aggiornato il valore di `ownerHasExpiredSales` con il risultato dell'operazione "OR" tra il vecchio valore della variabile e ed il controllo della vendita scaduta e posseduta dall'utente. Questa variabile, infatti, avrà valore `true` se l'utente possiede almeno una vendita scaduta, `false` altrimenti. Infine i tre array ed il valore boolean vengono assegnati alle rispettive variabili di stato.

4.5.2 Pop-up delle vendite scadute

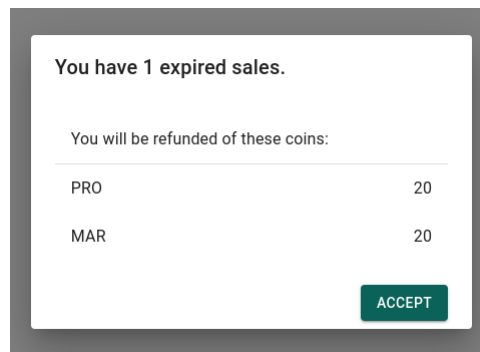


Figura 9: Pop-up mostrato se l'utente ha vendite scadute

Il pop-up in figura 9 viene visualizzato al caricamento della pagina nel caso l'utente abbia vendite scadute, ossia nel caso `ownerHasExpiredSales`, il cui valore è stato calcolato in 4.5.1, abbia valore `true`. Il componente custom che realizza il pop-up è `<SaleExpiredDialog>`. A questo viene passato attraverso la props `expiredSales` l'array `expiredSalesList`. Per realizzare la

finestra di dialogo, `<SaleExpiredDialog>` si basa sul componente `<Dialog>` di *Material-UI*. A quest'ultimo è stato inserito un titolo, una lista e un pulsante. Il titolo mostra all'utente il numero, ottenuto con la lunghezza dell'array passato nella props `expiredSales`, di vendite scadute che possiede. La lista elenca le monete e le quantità associate che l'utente può richiedere come rimborso. L'utente ottiene il rimborso premendo il pulsante "Accept".

4.5.2.1 Calcolo del rimborso

Al caricamento del componente `<SaleExpiredDialog>` viene richiamata la funzione `calcRefund()`.

```
1  const calcRefund = () => {  
2    const tmpRefundMap = new Map();  
3  
4    for(let sale of expiredSales) {  
5      for(let coin of sale.tokensOnSaleData) {  
6        tmpRefundMap.get(coin.symbol) ?  
7          tmpRefundMap.set(  
8            coin.symbol,  
9            parseInt(tmpRefundMap.get(coin.symbol)) + parseInt(coin.amount))  
10         :  
11         tmpRefundMap.set(coin.symbol, coin.amount);  
12      }  
13    }  
14  
15    setCoinsToRefund(Array.from(tmpRefundMap.entries()));  
16  }
```

Funzione `calcRefund()`

La funzione genera la Map `tmpRefundMap`, successivamente per ogni moneta in vendita di ogni vendita all'interno di `expiredSales`, ossia l'array contenente le vendite scadute dell'utente, controlla se l'indirizzo della moneta è

presente come chiave all'interno di `tmpRefundMap`. Se non è presente inserisce il simbolo come chiave e il la quantità della moneta in vendita come valore associato. Altrimenti aggiunge la quantità al valore già presente. Alla fine delle iterazioni, `tmpRefundMap` conterrà le monete e le loro quantità totali da rimborsare. Infine, `tmpRefundMap` viene convertito in array e assegnato a `coinsToRefund`, ossia l'array utilizzato dalla lista in `SaleExpiredDialog`.

4.5.2.2 Accettazione del rimborso

Premendo sul pulsante "Accept" viene richiamata la funzione `acceptRefund()`.

```
1 const acceptRefund = async () => {
2   const salesAddresses = [];
3
4   for(let sale of expiredSales) {
5     salesAddresses.push(sale.address);
6   }
7
8   const result = await saleCancelBatch(web3Instance, userAccount,
9     salesAddresses);
10
11   const alertMessage = result ? "Sales refunded successfully" : "Sales not
12     refunded. Something went wrong...";
13
14   setAlertState({
15     result: result,
16     message: alertMessage,
17     open: true,
18   });
19   props.onClose();
20 }
```

Funzione `acceptRefund()`

La funzione crea un array chiamato `salesAddresses` contenente tutti gli indirizzi delle vendite scadute contenute in `expiredSales`. Questo array viene poi passato come parametro alla chiamata della funzione `saleCancelBatch`. Viene, quindi, mostrato un messaggio che comunica all'utente il risultato dell'operazione e, infine, il pop-up viene chiuso.

4.5.3 Filtro delle vendite

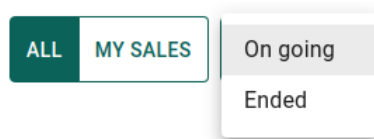


Figura 10: Filtro delle vendite

L'utente può filtrare la lista delle vendite per visualizzarle tutte oppure solo quelle che possiede. Per entrambi possono essere visualizzate quelle in corso oppure quelle terminate. Il filtro è composto da due pulsanti, realizzati con due `<ToggleButton>` contenuti all'interno di un `<ToggleButtonGroup>`, entrambi i componenti forniti da *Material-UI*. Questi componenti permettono che, alla pressione di un pulsante, venga graficamente selezionato il pulsante selezionato e vengano deselezionati gli altri. Il filtro è composto, inoltre, da un selettore, realizzato con il componente *Material-UI* `<Select>` contenente due `<MenuItem>` che rappresentano le due opzioni possibili. Premendo un `<ToggleButton>` viene richiamata la funzione `handleChangeFilter()`.

```
1 const handleChangeFilter = (event, filter) => {  
2   if(filter !== null) {  
3     setSelectedFilter(filter);  
4  
5     changeShowingSales(filter, isEndedStatusSelected);  
6   }  
}
```

```
7 }
```

Funzione handleChangeFilter()

La funzione ha come parametro `filter` che rappresenta l'indice del filtro selezionato, questo viene assegnato alla variabile `selectedFilter`. Viene infine chiamato il metodo `changeShowingSales()` passando il parametro `filter` e la variabile `isEndedStatusSelected`.

Selezionando, invece, un'opzione di `<Select>` viene chiamata la funzione `handleSalesStatusSelection()`.

```
1 const handleSalesStatusSelection = (status) => {  
2   setLoadingSaleList(true);  
3   const statusEnded = status === "Ended" ? true : false;  
4   setIsEndedStatusSelected(statusEnded);  
5   changeShowingSales(selectedFilter, statusEnded);  
6   setLoadingSaleList(false);  
7 }
```

Funzione handleSalesStatusSelection()

Questa funzione prende come parametro `status` che è una stringa che contiene il valore *"On going"* se è stato selezionato di visualizzare le vendite in corso, *"Ended"* altrimenti. Viene assegnato a `isEndedStatusSelected` il valore `true` se `status` ha valore *"Ended"*, altrimenti il valore `false`. Infine, anche questa funzione richiama `changeShowingSales()` passando `selectedFilter` e il valore assegnato a `isEndedStatusSelected`. Entrambe le funzioni descritte utilizzano il metodo `changeShowingSales()`, questo ha il compito di cambiare, in base ai filtri, la lista visualizzata dall'utente.

```

1  const changeShowingSales = (selectedFilter, salesStatus) => {
2    setLoadingSaleList(true);
3
4    const toFilterList = salesStatus ? endedSalesList : onGoingSalesList;
5
6    if(selectedFilter === 0) {
7      setShowingSalesList(toFilterList);
8    }
9
10   if(selectedFilter === 1) {
11     const filteredSales = toFilterList.filter((sale) => {
12       if(sale.owner === userAccount) return true;
13       return false;
14     });
15
16     setShowingSalesList(filteredSales);
17   }
18
19   setLoadingSaleList(false);
20 }

```

Funzione changeShowingSales()

changeShowingSales() ha due parametri: **selectedFilter**, ossia il filtro selezionato, e **salesStatus**, ossia lo stato delle vendite selezionato. In base al valore di **salesStatus**, viene assegnato all'array **toFilterList** la lista **endedSalesList**, se si vuole visualizzare le vendite terminate, altrimenti **onGoingSalesList**. In caso di **selectedFilter** con valore 0, ossia è stato selezionato di visualizzare tutte le vendite, viene assegnato **toFilterList** a **showingSalesList**. Altrimenti, se ha valore 1, vengono filtrate le vendite in **toFilterList** assegnando a **setShowingSalesList** solo quelle possedute dall'utente.

4.5.4 Rappresentazione grafica di una vendita

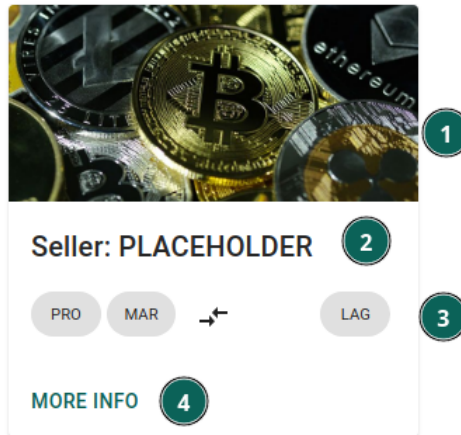
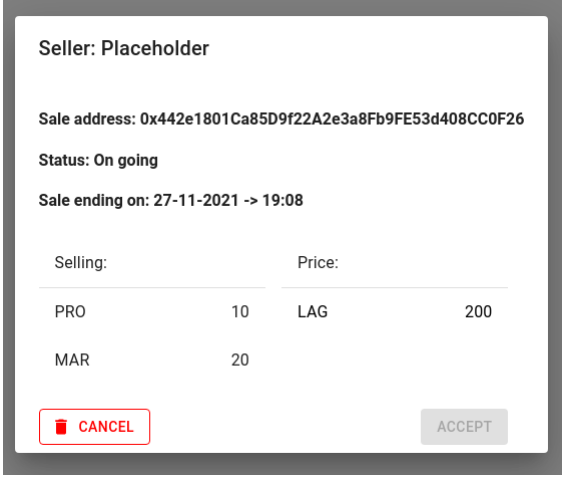


Figura 11: Informazioni di base di una vendita

Nell'immagine 12 è rappresentata l'interfaccia di una vendita della lista presente nella pagina "Sales List". Questa presenta alcune informazioni di base della vendita. Al punto 1 è illustrata un'immagine stock in tema di criptomonete. Al punto 2 è presente il nome del proprietario della vendita, questa funzionalità è ancora incompleta in quanto è prima necessaria l'integrazione con l'applicazione di CommonsHood. Al punto 3 sono rappresentate, nella colonna a sinistra, le monete in vendita e, nella colonna a destra, le monete accettate come pagamento dalla vendita. Infine al punto 4 è presente un pulsante che, alla pressione, apre una finestra di dialogo in cui l'utente può interagire con la vendita.

4.5.5 Finestra di dialogo di una vendita



Seller: Placeholder

Sale address: 0x442e1801Ca85D9f22A2e3a8Fb9FE53d408CC0F26

Status: On going

Sale ending on: 27-11-2021 -> 19:08

Selling:		Price:	
PRO	10	LAG	200
MAR	20		

Figura 12: Informazioni di base di una vendita

Come descritto in 4.5.4, premendo il pulsante "*More Info*" di una vendita viene aperta una finestra di dialogo. Questa mostra maggiori informazioni sulla vendita. Le informazioni visualizzate sono: il nome del venditore, che, come descritto in 4.5.4, non è ancora implementato; l'indirizzo della vendita, lo stato della vendita e la data di scadenza della vendita. La finestra contiene, inoltre, le liste delle monete in vendita e quelle come pagamento e le loro quantità. In fondo sono presenti due pulsanti. Il primo, colorato di rosso, permette all'utente di eliminare la vendita. Questo pulsante non viene mostrato in caso la vendita non appartenga all'utente. Il secondo pulsante, colorato di verde, permette all'utente di accettare la vendita. Questo è disabilitato in caso la vendita appartenga all'utente oppure in caso l'utente non abbia abbastanza monete per il pagamento. In caso la vendita sia già terminata, entrambi i pulsanti non vengono visualizzati.

4.5.5.1 Eliminazione della vendita

Premendo il pulsante di eliminazione della vendita viene chiamata la funzione `handleCancelSale()`.

```
1 const handleCancelSale = async () => {  
2   const cancelResult = await saleCancel(web3Instance, userAccount,  
    saleAddress);  
3  
4   const cancelMessage = cancelResult ? "Sale cancelled successfully" : "  
    Sale was not cancelled. Something went wrong...";  
5  
6   setAlertState({  
7     result: cancelResult,  
8     message: cancelMessage,  
9     open: true,  
10  })  
11  
12  props.onSaleOperation();  
13  
14  setShouldClose(true);  
15 }
```

Funzione `handleCancelSale()`

La funzione richiama il metodo `saleCancel()` passando `saleAddress`, ossia l'indirizzo della vendita, come parametro. Successivamente mostra un messaggio di risposta all'utente ed infine chiude la finestra di dialogo.

4.5.5.2 Accettazione della vendita

Premendo il pulsante per accettare la vendita viene chiamata la funzione `handleAcceptSale()`.


```

1  const handleAcceptSale = async () => {
2      const acceptResult = await saleAccept(web3Instance, userAccount,
        saleAddress, coinsToAccept);
3
4      const acceptMessage = acceptResult ? "Sale accepted successfully" : "
        Sale was not accepted. Something went wrong...";
5
6      console.log("SALE ACCEPT: ", acceptResult);
7      setAlertState({
8          result: acceptResult,
9          message: acceptMessage,
10         open: true,
11     });
12
13     props.onSaleOperation();
14
15     setShouldClose(true);
16 }

```

Funzione handleAcceptSale()

La funzione esegue la chiamata al metodo `saleAccept()`, funzione importata dal file `sale.js`, passando come parametri `saleAddress`, ossia l'indirizzo della vendita, e `coinsToAccept`, ovvero la lista delle monete che la vendita necessita come pagamento. Infine mostra all'utente il risultato dell'accettazione e chiude la finestra.

5 Conclusioni

Riferimenti bibliografici

- [1] Amazon. *What is Ethereum?* URL: <https://aws.amazon.com/it/blockchain/what-is-ethereum/>.
- [2] Andrew Carroll. *The art world needs blockchain*. URL: <https://irishtechnews.ie/the-art-world-needs-blockchain/>.
- [3] William Entriken et al. *EIP-721: Non-Fungible Token Standard*. URL: <https://eips.ethereum.org/EIPS/eip-721>.
- [4] Ethereum.org. *Ethereum Docs*. URL: <https://ethereum.org/en/developers/docs/standards/tokens/erc-20/>.
- [5] Ethereum.org. *Ethereum Docs - Types*. URL: <https://docs.soliditylang.org/en/v0.4.24/types.html>.
- [6] Ethereum.org. *Ethereum Docs - Visibility and getters*. URL: <https://docs.soliditylang.org/en/v0.4.24/contracts.html#visibility-and-getters>.
- [7] Ethereum.org. *Ethereum Docs - web3.eth.Contract - methods.myMethod.call*. URL: <https://web3js.readthedocs.io/en/v1.2.11/web3-eth-contract.html#methods-mymethod-call>.
- [8] Ethereum.org. *Ethereum Docs - web3.eth.Contract - methods.myMethod.send*. URL: <https://web3js.readthedocs.io/en/v1.2.11/web3-eth-contract.html#methods-mymethod-send>.
- [9] Ethereum.org. *Non-fungible tokens (NFT)*. URL: <https://ethereum.org/en/nft/>.
- [10] IBM. *What are smart contracts on blockchain?* URL: <https://www.ibm.com/topics/smart-contracts>.

- [11] Metamask. *Metamask docs*. URL: <https://docs.metamask.io/guide/>.
- [12] Mui-org. *Material-UI docs*. URL: <https://v4.mui.com/customization/components/>.
- [13] OpenZeppelin. *OpenZeppelin docs*. URL: <https://docs.openzeppelin.com/contracts/4.x/>.
- [14] Jimi S. *Blockchain: What are nodes and masternodes?* URL: <https://medium.com/coinmonks/blockchain-what-is-a-node-or-masternode-and-what-does-it-do-4d9a4200938f>.
- [15] Truffle suite. *Truffle suite docs*. URL: <https://www.trufflesuite.com/docs>.
- [16] Truffle suite. *Truffle suite docs - testing*. URL: <https://trufflesuite.com/docs/truffle/testing/testing-your-contracts>.
- [17] Truffle suite. *Truffle suite docs - Truffle Teams - Connecting to a sandbox*. URL: <https://www.trufflesuite.com/docs/teams/contract-manager/connecting-to-a-sandbox>.
- [18] UTBI. *Commonshood Dapp Extension Boilerplate*. URL: <https://gitlab.di.unito.it/utbi/commonshood-dapp-extension-boilerplate>.
- [19] Giuseppe Vanni. *Blockchain: cos'è, come funziona, tecnologia e applicazioni*. URL: <https://www.punto-informatico.it/blockchain-spiegazione/#h222751-0>.
- [20] Fabian Vogelsteller e Vitalik Buterin. *EIP-20: Token Standard*. URL: <https://eips.ethereum.org/EIPS/eip-20>.