Indice

1 Background

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 altro asset che gli utenti possono creare e possedere sono i coupon, questi sono rappresentati da token non fungibili basati sullo standard ERC-721, in questo modo ogni singolo coupon è univoco e diverso dagli altri. Gli utenti possono utilizzare i coupon per ottenere prestazioni e 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 *nodi*, 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.

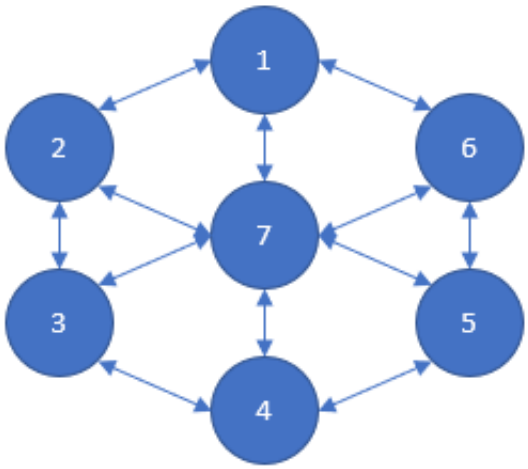

Figura 1: Rete di nodi

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[18].

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 specificazioni stabilite dagli sviluppatori della rete. I token possono essere di

tipi di differenti, a seconda delle specificazioni 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 specificazioni indicate nell'EIP-20[1] 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[19].

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

[1] Ethereum Improvement Proposal

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:

- `balanceOf(address _owner)`: restituise 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'esterno 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 uin256. 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 `uin256`. 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 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:

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

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`:

```
function Convenevoli() {
  return (
    <div>
      <Saluto nome="Sara" />
      <Saluto nome="Cahal" />
      <Saluto nome="Edite" />
    </div>
  );
}
```

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:

```
function Contatore() {
  const [contatore, setContatore] = useState(0);
  return (
    <div>
      <p>Hai cliccato {contatore} volte</p>
      <button
        onClick={() => setContatore(contatore + 1)}>
        Cliccami
      </button>
    </div>
  );
}
```

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()`.

```
function ContatoreConTitolo() {
  const [contatore, setContatore] = useState(0);

  useEffect(() => {
    document.title = `Hai cliccato ${contatore} volte`;
  });

  return (
    <div>
      <p>Hai cliccato {contatore} volte</p>
      <button
        onClick={() => setContatore(contatore + 1)}>
        Cliccami
      </button>
    </div>
  );
}
```

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>`:

```
.Button {
  width: "100px",
  height: "100px"
}

<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 ABI[3] 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

[2]oppure truffle migrate
[3]Application Binary Interface

sarebbero difficilmente comprensibili. Un' ABI, quindi, descrive le funzioni 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 `newtorks`, 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("<Funzionalita_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 Scrittura e test dei smart contracts

3 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.

3.1 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`.

3.1.1 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 documentazion 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`".

3.2 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`.

3.2.1 saleCreate

```
export const saleCreate = async (web3, sellerAddress,
    coinsOnSaleAddr, amountsOnSale, coinsToAcceptAddr,
    amountsToAccept, expirationDate) => {
  try {
    for(let i = 0; i < coinsOnSaleAddr.length; i++) {
      const coinInstance = new web3.eth.Contract(
        SMART_CONTRACTS.TKN_TMPLT_ABI,
        coinsOnSaleAddr[i],
      );

      await coinInstance.methods.approve(
        SMART_CONTRACTS.SALE_FCTRY_ADDR,
        amountsOnSale[i]
      ).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 contraact `SaleFactory`. Que-

sto viene effettuato creando un nuovo ogggetto 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.

3.2.2 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().call({ from: accountAddress, gasPrice: "0"});
24
25          let tokensOnSaleData = [];
26          let tokensToAcceptData = [];
27          const saleOwner = saleInfo[0];
28          const saleEnded = saleInfo[1];
29          const saleExpiration = saleInfo[6];
30
```

```
31        const tokensOnSaleAddr = saleInfo[2];
32        const amountsOnSale = saleInfo[3];
33        for(let i = 0; i < tokensOnSaleAddr.length; i++) {
34          const tokenData = await coinGetFullData(web3,
    accountAddress, tokensOnSaleAddr[i]);
35          tokenData['address'] = tokensOnSaleAddr[i];
36          tokenData['amount'] = amountsOnSale[i];
37          tokensOnSaleData.push(tokenData);
38        }
39
40        const tokensToAcceptAddr = saleInfo[4];
41        const amountsToAccept = saleInfo[5];
42        for(let i = 0; i < tokensToAcceptAddr.length; i++) {
43          const tokenData = await coinGetFullData(web3,
    accountAddress, tokensToAcceptAddr[i]);
44          tokenData['address'] = tokensToAcceptAddr[i];
45          tokenData['amount'] = amountsToAccept[i];
46          tokensToAcceptData.push(tokenData);
47        }
48
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 sucessive.
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 paramentro. 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.

3.2.3 saleAccept

```
export const saleAccept = async (web3, accountAddress,
    saleAddress, coinsToAccept) => {
  try {
    for(let coin of coinsToAccept) {
      const {address: coinAddress, symbol, amount} = coin;

      const tokenInstance = new web3.eth.Contract(
        SMART_CONTRACTS.TKN_TMPLT_ABI,
        coinAddress,
      );

      await tokenInstance.methods.approve(saleAddress, amount).
    send({from: accountAddress, gasPrice: '0'});
    }

    const saleInstance = new web3.eth.Contract(
      SMART_CONTRACTS.SALE_TMPLT_ABI,
      saleAddress,
    );
    const res = await saleInstance.methods.acceptSale().send({
    from: accountAddress, gasPrice: '0'});
    return true;
  } catch(error) {
    for(let i = 0; i < coinsToAccept.length; i++) {
      const coinInstance = new web3.eth.Contract(
        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 informazione 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'accettamento della vendita. Viene, quindi, resituito 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.

3.2.4 saleCancel

```
export const saleCancel = async (web3, accountAddress,
    saleAddress) => {
try {
    const saleInstance = new web3.eth.Contract(
      SMART_CONTRACTS.SALE_TMPLT_ABI,
      saleAddress,
    );

    const res = await saleInstance.methods.cancelSale().send({
    from: accountAddress, gasPrice: '0'});
    return true;
  } catch(error) {
    return false;
  }
}
```

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`.

3.2.5 saleCancelBatch

```
export const saleCancelBatch = async (web3, accountAddress,
    saleAddresses) => {
  try {
    const SaleFactoryInstance = new web3.eth.Contract(
      SMART_CONTRACTS.SALE_FCTRY_ABI,
      SMART_CONTRACTS.SALE_FCTRY_ADDR,
    );

    const res = await SaleFactoryInstance.methods.
    cancelBatchSales(saleAddresses).send({from: accountAddress,
     gasPrice: '0'});
    return true;
  } catch(error) {
    return false;
  }
}
```

Funzione saleCancelBatch

La funzione `saleCancel` permette di cancellare multiple vendite. Prende in input l'istanza di `web3`, l'indirizzo del proprieario 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`.

3.3 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 2.

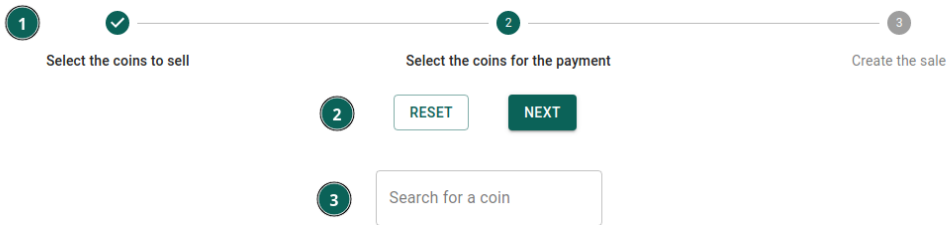

Figura 2: Componenti comuni

Per mostrare graficamente il progresso nei diversi step è presente, in cima alla pagina, un componente `<stepper>`, questo, come mostrato nell'immagine 2 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.

3.3.1 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

3.3.2 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.

3.3.3 Step 1: scelta dei token da vendere

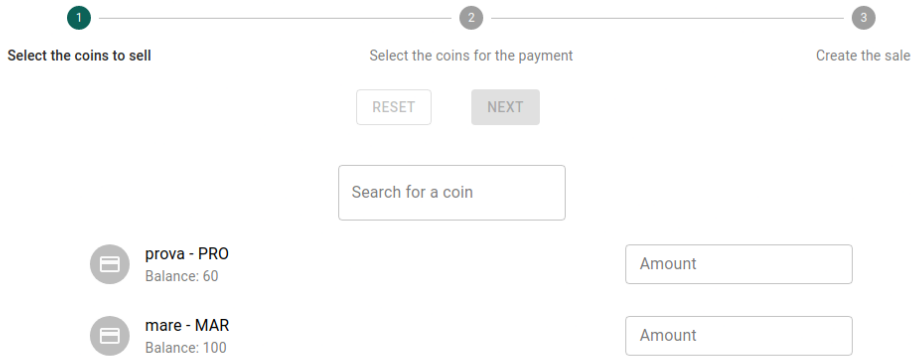

Figura 3: Pagina per la scelta dei token in vendita

La pagina contiene una lista, creata usando il componente `<List>`, ed una barra di ricerca, creata usando il componente `<TextField>`.
Ogni elemento della lista contiene le informazioni di un token posseduto dall'utente e un componente `<TextField>`, quest'ultimo permette l'inserimento della quantità desiderata del token da mettere in vendita. Gli eventi a cui la pagina reagisce possono essere divisi in tre categorie: caricamento della pagina, inserimento di quantità dei token e inserimento di testo nella barra di ricerca.

3.3.3.1 Caricamento della pagina

Al caricamento della pagina vengono istanziate diverse variabili che gestiscono lo stato della stessa.

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] Giuseppe Vanni. *Blockchain: cos'è, come funziona, tecnologia e applicazioni*. URL: https://www.punto-informatico.it/blockchain-spiegazione/#h222751-0.

[19] Fabian Vogelsteller e Vitalik Buterin. *EIP-20: Token Standard*. URL: https://eips.ethereum.org/EIPS/eip-20.