Blockchain and Cryptocurrencies

Corso di laurea Magistrale in Informatica

Università di Bologna

A. Y. 2020-2021

Ulderico Vagnoni 0000953964

8 settembre 2022

1 Obiettivi

La seguente è una documentazione esplicativa del progetto relativo al corso Blockchain and Cryptocurrencies tenuto dal Professor Stefano Ferretti presso l'università di Bologna.
L'obiettivo del progetto è quello di realizzare un'applicazione in ambiente ReactJS e Solidity, utilizzata per l'acquisto di prodotti made in Italy attraverso l'utilizzo di criptovalute (Ethers in questo caso) e per lo scambio di denaro, sempre sotto forma di criptovalute.
L'utente, una volta entrato nell'applicazione, dovrà connettere il suo portafoglio di MetaMask prima di accedere alle varie funzionalità. Una volta collegato, l'utente accede a varie componenti attraverso una NavBar:

- Marketplace: da qui è possibile visitare lo store, contente tutti prodotti acquistabili.
- Exchange: dove sarà possibile avviare lo scambio di denaro utilizzando l'indirizzo del destinatario.
- Claim: da qui, una volta effettuato l'acquisto, l'utente comunica la ricezione del prodotto.
- Purchased: contentente tutti gli acquisti effettuati nella piattaforma in attesa di essere confermati.

Ogni volta che l'utente effettua un acquisto, dal suo conto viene prelevata una quantità di denaro pari al doppio del prezzo del prodotto per motivi cautelari, infatti, una volta che l'utente riceve il suo prodotto, dopo aver comunicato al sito la corretta ricezione, metà della somma spesa andrà nel portafoglio del proprietario del sito, e l'altra meta tornerà nel portafoglio dell'utente.
Inoltre, a ogni acquisto viene associato un codice randomico, il quale può essere visionato dalla componente *Purchased*. Questo deve essere inserito nella componente *Claim* ogni volta che un prodotto arriva a destinazione, al fine di riprendere il denaro. Quando l'utente inserisce il codice randomico deve essere collegato con l'account MetaMask relativo all'acquisto, altrimenti tale operazione verrà negata.

Eventuali problematiche relative alla privacy sono state ignorate, essendo tale progetto illustrativo e venendo eseguito in locale.

2 Tecnologie utilizzate

Per la realizzazione di questo progetto sono state utilzzate varie tecnologie, queste sono:

- ReactJS: utilizzato per la realizzazione dell'interfaccia web.
- Ether.js: una libreria JavaScript che permette la comunicazione dell'applicazione web con la blockchain di Ethereum.
- Solidity: utilizzato per la creazione dello Smart Contract contenente tutte le funzioni utilizzate nel progetto.
- MetaMask: utilizzato per la creazione di wallet con i quali gestire il flusso di denaro entrante e uscente (in criptovalute).
- HardHat: utilizzato per la compilazione e il deploy dello Smart Contract.

3 ReactJS

React è un framework creato da Facebook utilizzato per creare applicazioni Javascript single-page mediante l'utilizzo di componenti. Il vantaggio quindi è di poter creare complesse interfacce web senza la necessità di ricaricare la pagina più volte. Il tutto avviene mediante l'utilizzo di *componenti*, ossia dei blocchi di codice scritti in JSX (JavaScript XML) renderizzati nella pagina web. Inoltre in questo progetto è stato utilizzato il tool Vite. Per la creazione dell'applicazione è stato utilizzato il comando:

```
npm init vite@latest
```

Una volta eseguito, tale comando genera una folder contente tutti i file necessari. Per eseguire l'applicazione in localhost viene utilizzato il comando da terminale:

```
npm run dev
```

Il quale esegue l'applicazione sulla porta 3000. All'interno della cartella *src* gli elementi più importanti sono:

- components: la cartella contenente tutte le componenti utilizzati all'interno del progetto.
- App.jsx: la componente base che contiene tutte le altre componenti dell'applicazione.
- main.jsx: dove si indica il root id all'index html e dove vengono renderizzate tutte le componenti.

Le componenti realizzate sono relative a tutti i servizi offerti nell'applicazione, quindi per lo scambio di denaro viene utilizzata una componente chiamata **Exchange**, mentre per l'acquisto viene utilizzata un'altra componente chiamata **Buy**. Servizi come l'acquisto o la visualizzazione degli ordini fanno utilizzo di due componenti differenti, una per il servizio in sé e l'altra per la generazione di *Card* con le quali visualizzare le informazioni e inviare una richiesta allo Smart Contract.

Al fine di utilizzare variabili globalmente, o passarle da una componente all'altra, è stato utilizzato un *context*, chiamato **transactionContext**. All'interno di questo *context*, oltre a esportare variabili tra diverse componenti, sono state anche implementate le funzioni necessarie al funzionamento dell'applicazione, come per esempio la connessione dell'account MetaMask o le funzioni necessarie a effettuare l'acquisto.

3.1 Implementazione

All'interno di questa sezione verrà descritto in che modo i servizi e le funzionalità precedentemente descritte sono state implementate. Per l'acquisto di un prodotto sono state utilizzate due componenti:

- Card.jsx: all'interno di questa componente viene descritta una carta relativa al prodotto. In questa componente viene definito una **const** *Input* la quale viene utilizzata per inserire l'indirizzo nel quale far spedire il prodotto attraverso la funzione **handleChange**, che modifica il valore della variabile di stato *value* (l'indirizzo appunto) e chiama la funzione **buyProduct** con la quale viene registrato l'acquisto sulla blockchain. Inoltre, la funzione **handleChange** viene chiamata ogni volta che il valore viene modificato.

```
const Input = ({placeholder, value, handleChange}) => (
    <input
        placeholder={placeholder}
        type="text"
        step="0.0001"
        value={value}
        onChange={(e) => handleChange(e, name)}
        className="my-2 w-full rounded-sm p-2 outline-none bg-transparent text-black border-none text-sm blue-glassmorphism"
    ></input>
);

-----------------------------------------------------------------

const handleChange = (e) => {
    setValue(() => (e.target.value));
}

-----------------------------------------------------------------
const handleSubmit = async (e) => {
    if(!value){
        alert("Insert your shipping address before!");
    }else{

        e.preventDefault();
        const randomIdentifier = Math.floor(Math.random() * 1000001);
        buyProduct(randomIdentifier, value, connectedAccount, props.title, props.price);
    }
}
```

```

```

La card viene generata in HTML come:

```
        <div className='card text-center'>
            <div className='overflow'>
                <img src= {props.img} className='card__image'/>
            </div>
            <div className='card-boy text-dark'>
                <h4 className='card-title'> {props.title} </h4>
                <p className='card-text text-secondary'> {props.description} </p>
            </div>
            <br></br>
            <p className="flex-col font-bold text-black text-base">
                    Insert your shipping address!
            </p>
            <Input placeholder="Address" name="address" type="text" handleChange={handleChange
    }/>
            <p className='border-[2px] hover:bg-[#e6560e] rounded-full font-bold'> Price (ETH):
     {props.price} </p>
            <div className='justify-center items-center flex'>
                {isLoading ? (
                    <></>
                ) : (
                    <button className='bg-[#933709] py-2 px-7 mx-4 rounded-full cursor-pointer
    hover:bg-[#e6560e] text-white bottom-0' type="button" onClick={handleSubmit}> Buy </button>
                )}
            </div>
        </div>

```

Tutte le informazioni come titolo, prezzo, e così via, vengono passate in input alla componente ogni volta che viene dichiarata.

- Buy.jsx: all'interno della componente Buy, vengono visualizzate le card relative ai prodotti. Le card vengono visualizzate a partire da un array chiamato *products*, il quale contiene tutte le informazioni necessarie che verranno poi mappate:

```


    const products = [
        {id: 1, url: '../../images/medium.png', name: 'Medium size moka', deescription: 'Coffe
    for 3', price: '0.00003'},
        {id: 1, url: '../../images/little.png', name: 'Little size moka', deescription: 'Coffe
    for 1', price: '0.00001'},
        {id: 1, url: '../../images/big.png', name: 'Big size moka', deescription: 'Coffe for
    4/5', price: '0.00005'},
    ]

    return (
        <>
        <div className="flex flex-col flex-1 items-center justify-start w-full md:mt-0 mt-10">
            <h2 className="text-1xl sm:text-3xl text-white py-1 justify-center">
                Remember: you will pay double the price, the money will be returned once your
    product arrives! <br />
            </h2>
            <div className="flex w-full justify-center items-center wrapper">
                {products.map((product) => (
                    <>
                    <div className="flex md:flex-row flex-col items-start justify-between md:p
    -20 py-12 px-4">
                        <Card img={product.url} title = {product.name} description = {product.
    deescription} price = {product.price}/>
                    </div>
                    </>
                ))}
            </div>
        </div>
        </>
    );


```

Le altre componenti realizzate fanno uso degli stessi costrutti, perciò si rimanda direttamente alla loro visione attraverso codice.

All'interno delle varie componenti si fa utilizzo di variabili con uno scope che va al di fuori della componente stessa, come per esempio le informazioni relative all'account MetaMask collegato, o le informazioni d'acquisto che dovranno

essere passate dalla componente Card allo Smart Contract. Perciò viene utilizzato un *context* all'interno del quale sono inserite queste variabili:

```
    const [connectedAccount, setCurrentAccount] = useState("");

    const [formData, setFormData] = useState({ address: '', amount: '', keyword: '', message: ''});

    .....
```

Queste vengono poi esportate mediante il seguente snippet:

```
    return (
        <transactionContext.Provider value={{connectWallet, isLoading, connectedAccount, formData,
    formDataPurchase, setFormData, setFormDataPurchase, handleChange, handleSubmitPurchase,
    sendTransaction, buyProduct, setIsloading, seeArrayProducts, arrayLenght, arrayProduct,
    productArrived, handleChangeClaim, formDataClaim}}>
            {children}
        </transactionContext.Provider>
    );
```

Inoltre vengono anche definite tutte le funzioni, come la connessione dell'account MetaMask:

```
    const {ethereum} = window;

    const connectWallet = async () => {
        try {
            if(!ethereum) return alert("Please install metamask");
            const accounts = await ethereum.request({method: 'eth_requestAccounts'});
            setCurrentAccount(accounts[0]);
            console.log(accounts[0]);
            window.location.reload();
        } catch (error) {
            console.log(error);
            throw new Error("No ethereum object")
        }
    }
```

O l'acquisto di un prodotto:

```
    const buyProduct = async (e1, e2, e3, e4, e5) => {
        try {
            if(!ethereum) return alert("Install metamask");
            const randomId = e1;
            const shippingAddress = e2;
            const buyersAddress = e3;
            const productName = e4;
            const price = e5;

            const addressContract = transactionContract.address;
            console.log(addressContract);
            console.log(randomId);

            const realPrice = (price*2).toString();
            const _price = (price * (10**18));

            const saveMemo = await transactionContract.purchaseProduct(randomId, shippingAddress,
    buyersAddress, productName, _price, {value: ethers.utils.parseUnits(realPrice, "ether"),
    gasLimit: 210000});
            setIsloading(true);
            await saveMemo.wait();

            setIsloading(false);
            console.log("Success");

            window.location.reload();


        } catch (error) {
            console.log(error);

        }
    }
```

In questo caso, vengono prese le informazioni ottenute dalla componente Card e viene effettuata una richieta attraverso la libreria Ether.js, la quale, specificando il prezzo del gas e il suo limite, chiama la funzione **purchaseProduct**, passando la corretta quantità di denaro. Una volta terminata l'operazione, verrà eseguito un refresh della pagina.

Il codice per l'invio di una transazione è analogo:

```
const sendTransaction = async () => {
    try {
        if(!ethereum) return alert("Install metamask");
        const {addressTo, amount, keyword, message } = formData;
        const transactionContract = getEthereumContract();
        const parsedAmount = ethers.utils.parseEther(amount);

        await ethereum.request({
            method: 'eth_sendTransaction',
            params: [{
                from: connectedAccount,
                to: addressTo,
                gas: '0x5208',
                gasLimit: 500000,
                value: parsedAmount._hex,
            }]
        })

        const transactionHash = await transactionContract.addToBlockchain(addressTo,
parsedAmount, message, keyword);

        setIsloading(true);
        console.log(isLoading);
        await transactionHash.wait();
        setIsloading(false);
        const transactionCount = await transactionContract.getTransactionCount();

        setTransactionCount(transactionCount.toNumber());


        window.location.reload();

    } catch (error) {
        console.log(error);

    }
}
```

Qui, al contrario dell'acquisto, l'invio di denaro all'utente desiderato avviene mediante una richiesta della libreria Ether.js e non mediante il codice dello Smart Contract.

Inoltre vengono anche richiamate le funzioni necessarie a visualizzare i prodotti acquistati nella sezione **Purchased** o per avviare la procedura di **Claim**.

4 Solidity

Solidity è un linguaggio di programmazione Object-Oriented di alto livello utilizzato per l'implementazione di Smart Contracts, ossia programmi che governano il comportamente di un account all'interno di Ethereum. Questo è molto simile al linguaggio C++, Python e JavaScript. All'interno di questo progetto è stato utilizzato per realizzare lo Smart Contract **Transaction.sol**, utilizzato per l'acquisto di prodotti, per lo scambio di denaro e per la visualizzazione delle transazioni precedentemente eseguite.

4.1 Implementazione

Le prime cosa che vengono fatte all'interno del contratto sono la definizione dell'inidirizzo del proprietario, in questo caso un secondo account creato su MetaMask, e la dichiarazione di un mapping chiamato **moneySent** rappresentante la quantità di denaro versata per un determinato ordine.

Inoltre, andando in ordine, i tre pincipali servizi implementati dallo Smart Contract sono:

- Trasferimento di denaro: ciò avviene mediante la funzione **addToBlockchain**, attraverso questa funzione vengono salvate tutte le informazioni del tracciamento di denaro attraverso la **struct** ***TransferStruct*** e viene generato un **Event** relativo. Lo spostamento di denaro vero e proprio, come detto in precedenza, avviene mediante la libreria Ether.js.

```
    uint256 transactionCounter;

    event Transfer(address from, address receiver, uint amount, string message, uint256
    timestamp, string keyword);

    struct TransferStruct {
        address sender;
```

```
        address receiver;
        uint amount;
        string message;
        uint256 timestamp;
        string keyword;
    }

    TransferStruct[] transactions;

    function addToBlockchain(address payable receiver, uint amount, string memory message,
    string memory keyword) public {
        transactionCounter += 1;
        transactions.push(TransferStruct(msg.sender, receiver, amount, message, block.timestamp
    , keyword));

        emit Transfer(msg.sender, receiver, amount, message, block.timestamp, keyword);
    }
    function getAllTransactions() public view returns (TransferStruct[] memory) {
        return transactions;
    }
    function getTransactionCount() public view returns(uint256) {
        return transactionCounter;
    }


```

- Acquisto di prodotti: l'acquisto e la conferma della recezione del prodotto avvengono mediante le funzioni **purchaseProduct** e **productArrived**. Attraverso la prima avviene l'acquisto, le informazioni a riguardo vengono salvate nell'array di struct **arrayPurchased**, e verrà memorizzata la quantità di denaro da reinviare all'utente in caso di arrivo del prodotto mediante **moneySent** (l'identificatore randomico viene generato tramite React):

```
    function purchaseProduct(uint _randomIdentifier, string memory _shippingAddress, address
    _buyerAddress, string memory _productName, uint _price) public payable {
        productBought memory val = productBought(_randomIdentifier, _shippingAddress,
    _buyerAddress, _productName);
        arrayPurchased.push(val);
        lenght += 1;
        moneySent[_randomIdentifier] = ((_price/2));
    }

```

Una volta che il prodotto viene ricevuto dall'utente, viene chiamata la funzione **productArrived**, questa, prendendo in input l'identificatore randomico e l'indirizzo del compratore, verifica che tale acquisto sia ancora in attesa di essere ricevuto. Se il prodotto non è ancora arrivato, allora verrà trasferita la quantità di denaro contenuta in **moneySent** sia all'utente orginario che al proprietario del contratto, andando a fissare a zero il valore contenuto dalla variabile. In caso contrario avviene un revert, in modo che gli Ethers consumati per la transazione tornino al proprietario:

```

    function productArrived(uint _randomIdentifier, address payable _buyerAddress) public
    payable {
        bool isThere = false;
        for(uint i = 0; i < lenght; i++){
            if((keccak256(abi.encodePacked(_buyerAddress)) == keccak256(abi.encodePacked(
    arrayPurchased[i].buyerAddress))) && arrayPurchased[i].randomIdentifier ==
    _randomIdentifier){
                delete arrayPurchased[i];
                isThere = true;
            }
        }
        if(isThere){
            _buyerAddress.transfer(moneySent[_randomIdentifier]);
            payable(ownerAddress).transfer((moneySent[_randomIdentifier]*3));
            moneySent[_randomIdentifier] = 0;
        }else{
            revert();
        }
    }

```

- Tracciamento degli acquisti: il tracciamento degli acquisti infine avviene mediante l'utilizzo dell'array **arrayPurchased**, il quale è un'array di **struct** contenente informazioni quali l'identificatore randomico, l'indirizzo di spedizione, l'indirizzo del compratore e il nome del prodotto:

```

```

```
    struct productBought{
        uint randomIdentifier;
        string shippingAddress;
        address buyerAddress;
        string productName;
        //bool status;
    }


    function searchInArray(uint _randomIdentifier) public view returns(productBought memory) {
        for(uint i = 0; i <= lenght; i++){
            if(_randomIdentifier == arrayPurchased[i].randomIdentifier){
                return(arrayPurchased[i]);
            }
        }
        return(arrayPurchased[0]);
    }

    function seeLenght() public view returns(uint) {
        return(lenght);
    }

    function seeArray() public view returns(productBought[] memory) {
        return(arrayPurchased);
    }

```

4.2 Deploy con HardHat e testnet utilizzata

Il deploy del contratto è avvenuto mediante l'ambiente di sviluppo di Ethereum HardHat. Questo è avvenuto sulla testnet **Goerli**, in modo da poter effettuare debugging senza l'utilizzo di denaro vero.

Il file di configurazione di HardHat è:

```
require('@nomiclabs/hardhat-waffle');

module.exports = {
  solidity: '0.8.0',
  networks: {
    goerli: {
      url: 'https://eth-goerli.alchemyapi.io/v2/UASePshVPhfSvgf-SYsAZtqm0Zl8Ov6z',
      accounts: ['08760aac5e1005ef4bd2a7fb1ef45dff2194ddea63a1327541f32fb31d37721a']
    }
  }
}
```

Sempre all'interno del context **transactionContext** vengono caricati l'ABI e l'address del contratto a partire dal file JSON ottenuto a seguito del deploy del contratto.

5 MetaMask

Al fine di testare il corretto funzionamento dell'applicazione sono stati creati tre account MetaMask, uno rappresentante il generico utente che si interfaccia con l'applicazione, uno per un generico utente al quale inviare il denaro, e l'ultimo relativo al *proprietario* dell'applicazione, il quale ogni volta che un prodotto arriva a destinazione guadagnerà la somma pattuita.

6 Codice

Il codice può essere visualizzato nella seguente repository di **GitHub**: `https://github.com/Uderr/Blockchain`