

UNIVERSITÀ DEGLI STUDI DI NAPOLI “PARTHENOPE”
SCUOLA INTERDIPARTIMENTALE DELLE SCIENZE, DELL’INGEGNERIA E DELLA SALUTE
CORSO DI LAUREA IN INFORMATICA



PROGETTO DI RETI DI CALCOLATORI E LABORATORIO

Mooncake

DOCENTI

Aniello Casiglione

Alessio Ferone

CANDIDATI

Jacopo Gennaro

Esposito

0124001466

Alessandra Lorello

0124002024

Anno Accademico 2021-2022

Indice

1	Introduzione	3
1.1	Traccia Progetto	3
1.2	Perché "Mooncake"?	3
2	Descrizione schemi dell'architettura	4
2.1	Descrizione dei contratti	4
2.2	BlockChain	6
2.3	Mooncake	6
3	Descrizione e schemi del protocollo applicazione	7
3.1	Descrizione del protocollo applicazione	7
3.2	Schemi del protocollo	7
4	Dettagli implementativi	11
4.1	Smart Contract	11
4.2	ReceivePayment	11
4.2.1	ReceivePay()	11
4.2.2	PayWithBalance()	12
4.3	StorePaymentDiscount	13
4.3.1	Struct InfoPayment e array paymentArray	13
4.3.2	AddPayment()	14
4.3.3	SetBalance()	15
4.3.4	CheckBalance()	15
4.4	Webapp di Mooncake	15
4.4.1	Struttura della webapp Mooncake	16
4.4.2	Transazioni	16
5	Manuale Utente	18
5.1	Prerequisiti	18
5.2	Configurare Ganache	18
5.3	Installazione e configurazione di Mooncake	19
5.4	Login con Metamask	23
5.5	Attività sulla blockchain	25

Capitolo 1

Introduzione

1.1 Traccia Progetto

Si vuole progettare ed implementare una Dapp basata su tecnologia blockchain per gestire i pagamenti di un negozio virtuale.

La Dapp è composta da un'interfaccia web collegata a *Metamask* per i pagamenti ed uno smart contract che riceve i pagamenti ed applica uno sconto previsto dall'esercente.

L'ammontare dello sconto insieme all'indirizzo dell'acquirente vengono salvati in un secondo smart contract.

Un utente che ha dei fondi salvati nel secondo smart contract potrà utilizzarli per effettuare pagamenti al primo smart contract.

1.2 Perché "Mooncake"?

Prima di approfondire le specifiche di questo progetto verrà fornita una breve spiegazione dell'origine del nome "Mooncake". Si è deciso di utilizzare questo nome per due motivi principali. Il primo riguarda l'ambiente di sviluppo di Ethereum, all'interno del quale molti tool prendono il nome da pietanze dolci ed altri cibi, ad esempio "Truffle", "Ganache", "Mocha" e "Chai"; pertanto si è deciso di continuare questa golosa tradizione. Inoltre le mooncake, o tortini lunari, sono un popolare dolce di origini cinesi consumato durante la Festa di Metà Autunno, generalmente condiviso, esattamente come all'interno di una blockchain le informazioni e le transazioni sono distribuite fra tutti i nodi della stessa.

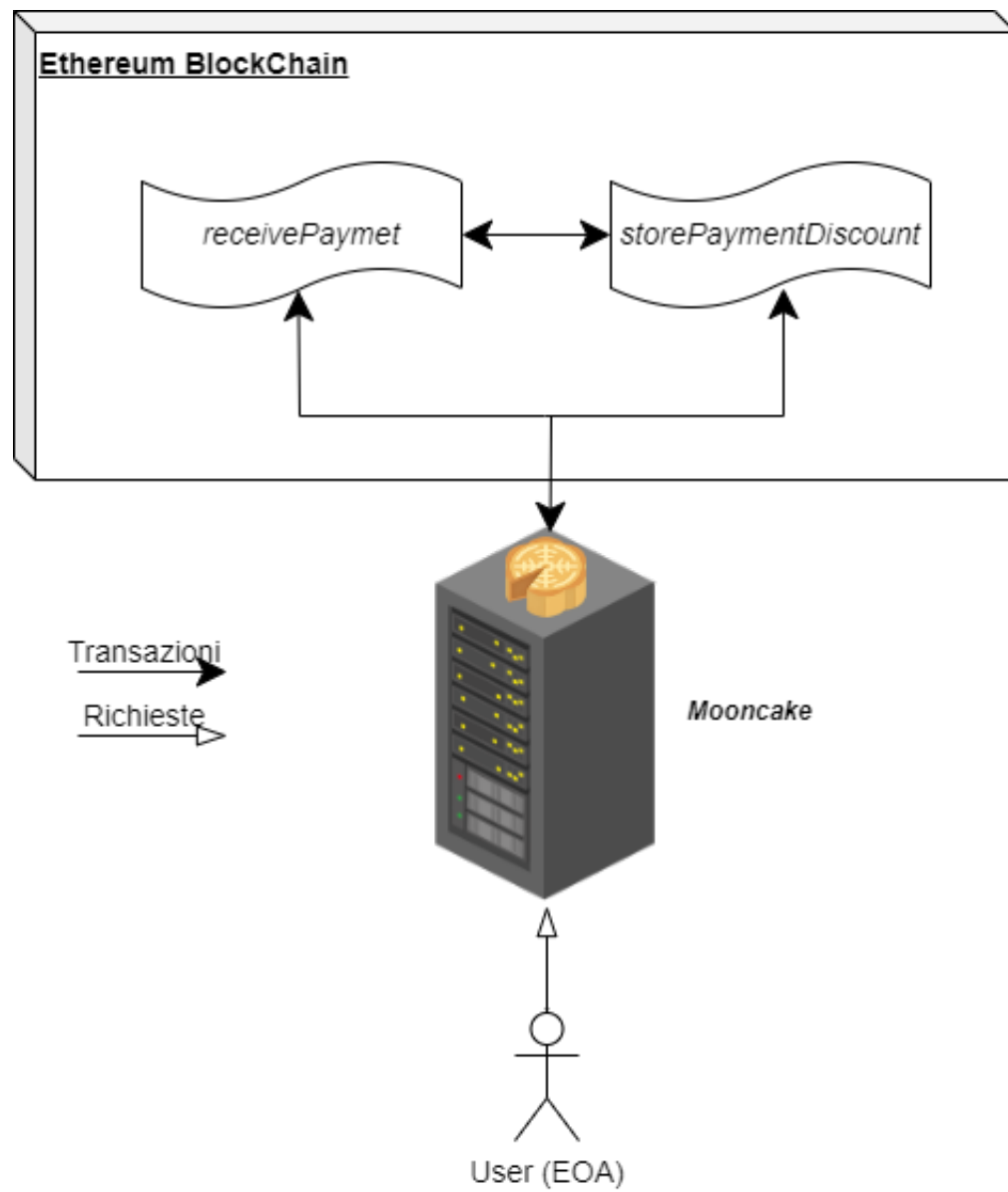
Capitolo 2

Descrizione schemi dell'architettura

2.1 Descrizione dei contratti

Questa Dapp è basata principalmente su due smart contracts:

- `receivePayment`:
 - `receivePay()`**: riceve il pagamento e ne calcola la percentuale di sconto da salvare nel saldo;
 - `payWithBalance()`**: permette il pagamento di un oggetto attraverso il saldo.
- `storePaymentDiscount`:
 - `addPayment()`**: crea un nuovo saldo o aggiorna uno preesistente;
 - `setBalance()`**: setta il saldo di un utente su un valore dato in input;
 - `checkBalance()`**: controlla il saldo di un utente tramite l'indirizzo del suo wallet.



2.2 BlockChain

In questo progetto viene utilizzata l'architettura della blockchain di Ethereum, volta all'implementazione di un'applicazione distribuita.

All'interno di una blockchain ogni nodo o peer possiede una copia di tutte le informazioni contenute all'interno della stessa. Ogni modifica effettuata attraverso una transazione deve essere convalidata dalla maggioranza dei nodi per essere approvata.

Questa caratteristica intrinseca della blockchain è perfetta per offrire maggiore *fault tollerance* e *sicurezza* rispetto ad un'architettura tradizionale. Ciò la rende adeguata per sviluppare un sistema di pagamento.

All'interno di una blockchain di Ethereum esistono due tipi di account: gli **Account Contract** e gli **Externally Owned Account**.

L'esecuzione di un contratto è causata dalla propagazione della transazione quando il destinatario di quest'ultima è il contratto stesso.

Quando un contratto riceve una transazione viene innescata la sua esecuzione nella *EVM (Ethereum Virtual Machine)*. Visto che un contract account non possiede una chiave privata, non può generare una transazione ex-novo, tuttavia possono reagire ad una transazione chiamando altri contratti o creando transazioni per trasferire Ether. All'interno della nostra Dapp si può trovare un esempio di queste due funzioni.

Nel momento in cui dobbiamo effettuare delle transazioni, lo smart contract, **receivePayment**, invocherà il secondo smart contract per memorizzare le informazioni relative al saldo, così da effettuare operazioni di controllo su di esso. Per quanto riguarda il trasferimento di Ether, verrà effettuato un rimborso del saldo quando lo si sfrutterà per pagare.

2.3 Mooncake

Mooncake, oltre ad essere composta da un backend basato sulla blockchain Ethereum, possiede un backend e un frontend minimale scritti entrambi in *Javascript*. Tramite questa webapp è possibile interagire con la blockchain ed effettuare le transazioni.

Capitolo 3

Descrizione e schemi del protocollo applicazione

3.1 Descrizione del protocollo applicazione

Il nostro protocollo applicazione prevede la presenza di un account utente tramite il quale può utilizzare il servizio fornito.

Attraverso l'applicazione, un utente può effettuare pagamenti in due modalità: quella di default e quella attraverso il saldo conservato nel proprio account.

Per offrire all'utente questi servizi vengono utilizzati i due smart contracts: **receivePayment** e **storePaymentDiscount**, descritti in precedenza.

3.2 Schemi del protocollo

In una blockchain Ethereum, per permettere lo scambio di messaggi fra gli utenti (compresi i contratti), vengono utilizzate delle transazioni.

Una transazione è composta da:

- **destinatario** del messaggio:
è colui a cui è diretto il messaggio, composto da un indirizzo di 20 byte. Può essere un Externally Owned Account oppure un contratto (contract account);
- firma del **mittente**:
ovvero l'indirizzo dell'account che invia la transazione. Come nel destinatario, può essere un EOA o un contract account;
- **valore** di Ether da trasferire:
ovvero la quantità di Ether inviati fra mittente e destinatario;
- un campo **dati** (opzionale):
viene utilizzato nelle *decentralized app* per definire un comportamento dei

contratti. L'invio di transazioni con dati permette l'utilizzo dei contratti in modo dinamico;

- un valore **gasLimit**:
ovvero il limite definito sul consumo di gas che l'esecuzione può generare. Dato che Ethereum definisce un sistema *Turing completo*, gli *halting problems* vengono risolti concludendo l'esecuzione al termine del gas;
- un valore **gasPrice**:
ovvero il prezzo in Ether corrispondente all'unità *gas*, basta moltiplicarlo al *gasLimit* per determinare il costo in Ether dell'intera esecuzione.

Il protocollo applicazione prevede due transazioni effettuate dall'utente (EOA) verso lo smart contract principale, chiamato **receivePayment**.

Nella transazione principale l'utente invia il numero di Ether necessari a finalizzare l'acquisto dell'oggetto. È compito di questo contratto controllare che la somma inviata corrisponda al prezzo dell'articolo, successivamente calcolerà la percentuale di sconto e invocherà lo smart contract **storePaymentDiscount** per memorizzare questa informazione nel saldo dell'utente.

La seconda transazione implementata consente ad un utente di pagare un oggetto attraverso il saldo disponibile.

Prima viene invocato il metodo **checkingBalance()** del secondo smart contract, per verificare che il saldo possa coprire, almeno in parte, il prezzo dell'articolo. In caso di esito positivo, il saldo viene resettato a zero e l'importo pagato corrispondente al saldo viene rimborsato mediante una transazione indirizzata all'utente.

Nell'altra casistica, invece, assistiamo ad un **revert()** della transazione e alla propagazione di un messaggio d'errore per fondi insufficienti.

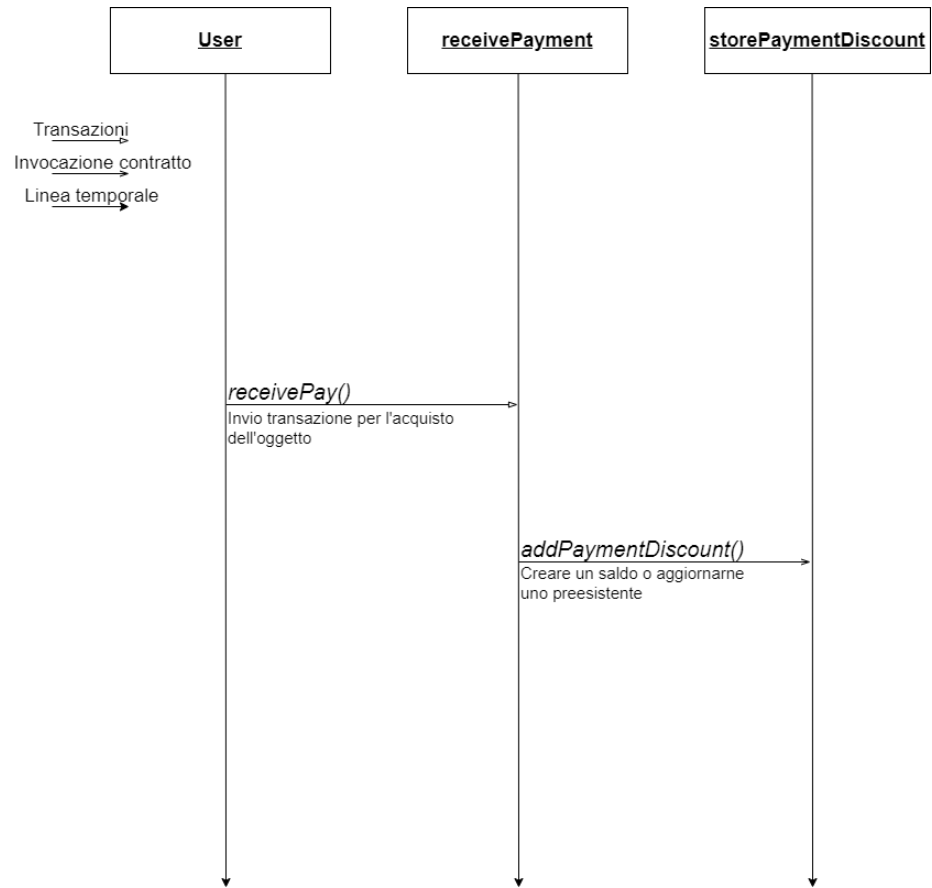


Figura 3.1: Transazione principale

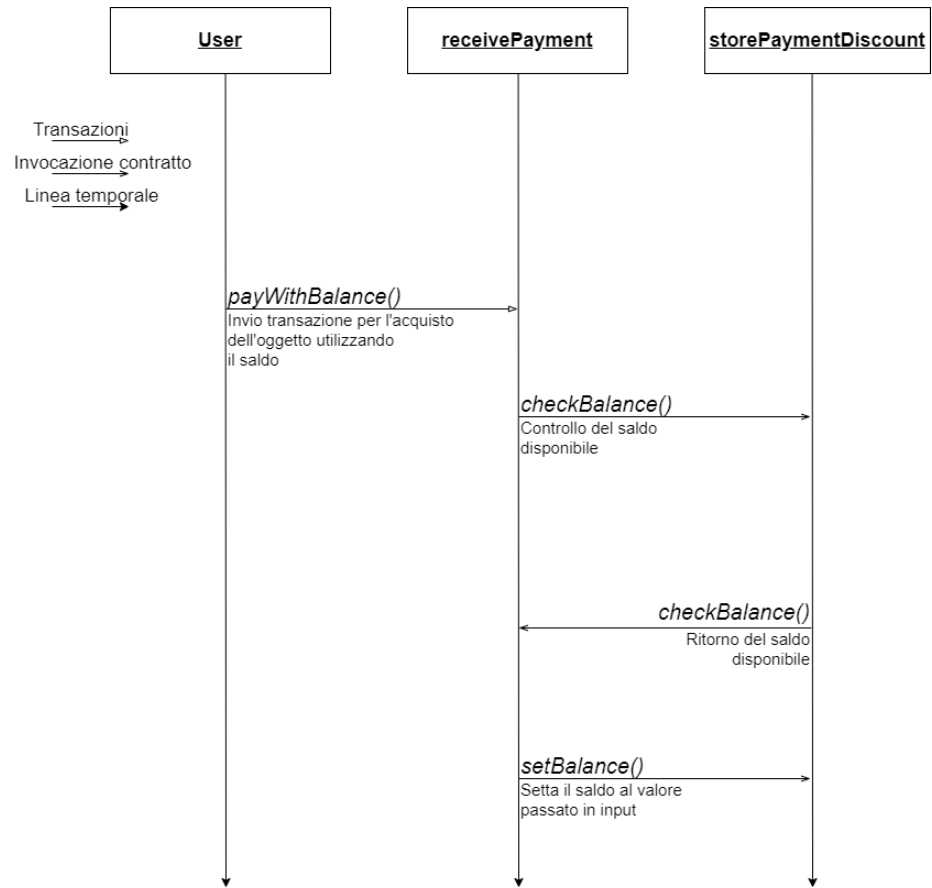


Figura 3.2: Transazione secondaria

Capitolo 4

Dettagli implementativi

4.1 Smart Contract

Gli smart contracts sono stati sviluppati con il linguaggio di programmazione *Solidity*, specializzato nella scrittura di smart contracts e fortemente influenzato da *C++*, *Javascript* e *Python*.

Questo linguaggio è sia compilato che interpretato, difatti, proprio come avviene in *Java*, vi è una precompilazione che genera un bytecode a partire da un file *".sol"*. Al momento dell'esecuzione il bytecode verrà dalla EVM.

4.2 ReceivePayment

ReceivePayment è il contratto principale, il quale si occupa di gestire i pagamenti ricevuti dagli utenti finali. Questo contratto è composto da due metodi principali, entrambi *payable*, in quanto devono poter essere invocati attraverso una transazione: **receivePay()** e **payWithBalance()**.

4.2.1 ReceivePay()

Il metodo *receivePay()* si occupa di gestire pagamenti per lo smart contract e accetta in input due parametri: *uint _price* e *address _t*.

Il primo parametro, *uint _price*, è un intero senza segno, utilizzato per definire il prezzo dell'articolo ed effettuare i controlli sullo stesso, in particolare utilizza il metodo **require()**. La comparazione fra il valore della transazione, **msg.value**, e il prezzo dell'articolo, *_price*, necessita che il prezzo venga prima convertito in *Wei*, in quanto la transazione viene effettuata utilizzando questa sottounità dell'Ether.

Successivamente, a partire dal prezzo, verrà calcolato lo sconto da salvare nel saldo e temporaneamente memorizzato nella variabile *uint discount*.

Il secondo parametro, *address _t*, è un indirizzo Ethereum, atto all'identificazione del secondo smart contract, che verrà utilizzato per creare un'istanza

di **storePaymentDiscount** attraverso il metodo **Existing()**. In seguito verrà invocato il metodo **addPayment()** di **storePaymentDiscount** che accetta in input due parametri: *uint discount* e *address idWallet*.

Il primo, *uint discount*, è lo sconto da applicare al saldo, mentre il secondo, *address idWallet*, è l'indirizzo pubblico dell'EOA mittente della transazione.

```
1 function receivePay(uint _price, address _t) public payable {
2     require(msg.value == _price*10**18, 'Need to send enough ether');
3     //Need to convert _price in wei before the comparison
4     discount = (_price*10)/100;
5     //Set the discount percentage
6     Existing(_t);
7     //Creating a new instance of the smart contract StorePaymentDiscount
8     sPD.addPayment(discount, msg.sender);
9     //Adding the information about the payment
10 }
```

4.2.2 PayWithBalance()

Il metodo *payWithBalance()* si occupa di gestire i pagamenti attraverso il saldo e accetta in input due parametri: *uint _price* e *address _t*.

Il secondo parametro *address _t*, così come in precedenza, è l'indirizzo del secondo smart contract, utilizzato per creare un'istanza di esso, attraverso il metodo **Existing()**.

Successivamente verrà chiamato il metodo **checkBalance()** di **storePaymentDiscount**, passandogli in input **msg.sender** ovvero l'indirizzo pubblico del mittente della transazione.

Il metodo **checkBalance()** restituirà il valore del saldo disponibile dell'account associato dell'indirizzo passatogli in input (verranno descritti in seguito i dettagli inerenti alle informazioni relative al saldo). Dopodichè viene effettuato un controllo per verificare che il saldo non sia vuoto, in caso di esito positivo verrà effettuato un **revert()** della transazione.

Il secondo controllo si occuperà di verificare che il saldo sia sufficiente a coprire in toto, o almeno in parte, il prezzo dell'articolo. In caso di esito positivo il saldo verrà settato a zero mediante il metodo **setBalance()** e l'ammontare del saldo precedente verrà restituito all'acquirente, mediante una transazione sotto forma di rimborso. In caso di esito negativo verrà effettuato un **revert()** della transazione.

```

1 function payWithBalance(uint _price, address _t) public payable{
2     Existing(_t);
3     uint _balance = sPD.checkBalance(msg.sender);
4     if(_balance == 0){
5         revert("Balance empty");
6     }
7     if((_price - _balance) >= 0){
8         //Check that the balance is enough
9         sPD.setBalance(msg.sender, 0);
10        //Set the balance to 0
11        payable(msg.sender).transfer(_balance*1 ether);
12        //Return the money saved to the EOA
13    }else{
14        revert("There are some problems with your balance");
15    }
16 }

```

4.3 StorePaymentDiscount

Questo contratto serve ad esporre dei metodi atti alla gestione del saldo e al controllo delle informazioni relative a quest'ultimo. È composto da tre metodi: **addPayment()**, **setBalance()** e **checkBalance()**.

Abbiamo una struttura **InfoPayment** e un array, **paymentArray**, di tipo **InfoPayment**.

4.3.1 Struct InfoPayment e array paymentArray

La struct **InfoPayment** viene utilizzata per descrivere le informazioni inerenti al saldo dell'utente, è composta da tre campi: il campo **address idwallet**, il campo **uint discount** e il campo **bool exists**.

Il campo **idwallet** corrisponde all'indirizzo dell'utente e viene utilizzato per identificare il saldo appartenente a quest'ultimo. Il campo **discount** serve a memorizzare il saldo dell'utente. Il campo **exists** è un valore *booleano* utilizzato nelle fasi successive per verificare l'esistenza di quella data struct con un certo **idwallet**.

Infine abbiamo un array, **paymentArray** di tipo **InfoPayment**, dove verranno conservati i singoli saldi relativi ai vari utenti.

```

1 struct InfoPayment {
2     address idwallet;
3     //Used for mapping
4     uint discount;
5     //Balance of the account
6     bool exists;
7     //Boolean value used to check if payment struct already exists
8 }
9 mapping(address => InfoPayment) public payments;
10 InfoPayment[] paymentArray;

```

4.3.2 AddPayment()

Il metodo *addPayment()* si occupa di creare o aggiornare le informazioni relative al saldo di un utente e accetta in input due parametri: *uint _discount*, *address _idwallet*.

Per prima cosa bisogna effettuare un controllo per verificare che non esista già un'istanza del saldo relativa a quell'utente all'interno di **paymentArray**. Nel caso in cui il campo **exists** sia *false* il saldo del nuovo utente, creato opportunamente, verrà aggiunto all'array dei pagamenti, **paymentArray**.

In caso contrario, viene effettuata una ricerca dell'**idwallet** all'interno del **paymentArray** per recuperare il saldo dell'utente, il quale verrà aggiornato aggiungendo il valore di **discount**.

```

1 function addPayment(uint _discount, address _idwallet) public {
2     if(!payments[_idwallet].exists){
3         //Checking if the balance is already present
4         payments[_idwallet] = InfoPayment(_idwallet, _discount, true);
5         //If not, create the balance
6         paymentArray.push(payments[_idwallet]);
7     }
8     else{
9         for(uint i = 0; i < paymentArray.length; i++){
10             if(paymentArray[i].idwallet == _idwallet){
11                 paymentArray[i].discount =
12                     paymentArray[i].discount + _discount;
13                 //Updating the balance
14             }
15         }
16     }
17 }
18 }

```

4.3.3 SetBalance()

Il metodo *setBalance()* si occupa di impostare il saldo ad un dato valore passato in input e accetta due parametri: **address** `_idwallet`, **uint** `_discount`. Innanzitutto bisogna recuperare all'interno dell'array, **paymentArray**, il saldo relativo a quell'utente attraverso l'indirizzo dell'utente, `_idwallet`, e successivamente settarlo al valore di `_discount`.

```
1 function setBalance(address _idwallet, uint _discount) public{
2     for(uint i = 0; i < paymentArray.length; i++){
3         if(paymentArray[i].idwallet == _idwallet){
4             paymentArray[i].discount = _discount;
5             //Set the balance to the value _discount
6         }
7     }
8 }
```

4.3.4 CheckBalance()

Il metodo *checkBalance()* si occupa di ritornare il saldo di un dato utente a partire dal suo indirizzo, accetta in input il parametro **address** `_idwallet` e ha come valore di ritorno **uint** `_discount`.

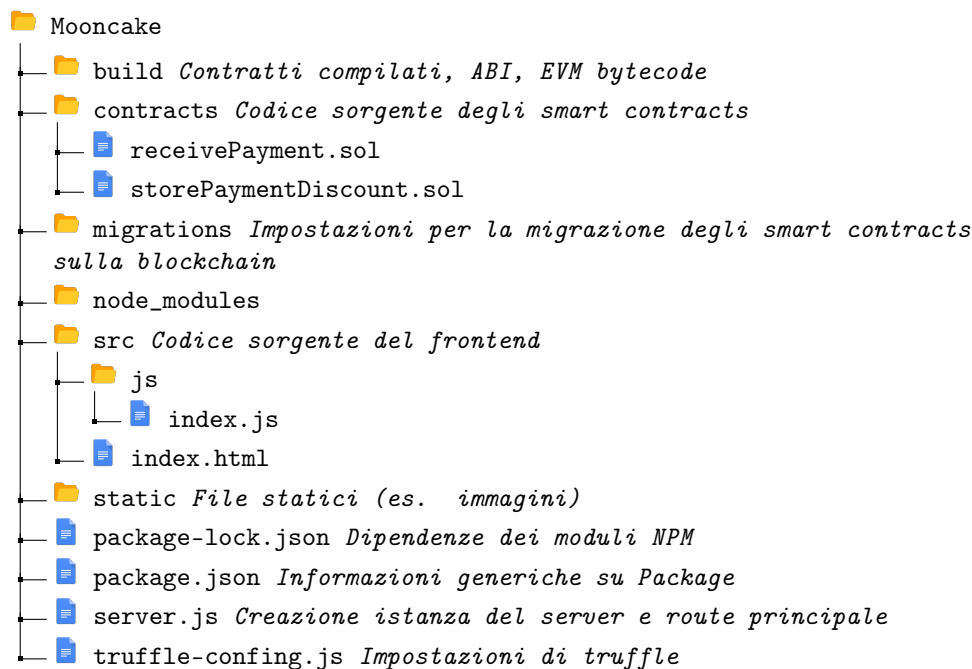
```
1 function checkBalance(address _idwallet) public view returns
2     (uint discount){
3     for(uint i = 0; i < paymentArray.length; i++){
4         if(paymentArray[i].idwallet == _idwallet){
5             return paymentArray[i].discount;
6         }
7     }
8 }
```

4.4 Webapp di Mooncake

La webapp di Mooncake è minimale, offre agli utenti la possibilità di effettuare le transazioni previste mediante l'uso di una comoda interfaccia grafica. È stata realizzata facendo prevalentemente affidamento su *Javascript*, il runtime system *Node.js* e la libreria *web3.js*, utilizzata per interfacciarsi con la blockchain. Oltre alle tecnologie sopracitate, per la parte grafica è stata utilizzata la libreria frontend *Bootstrap*.

4.4.1 Struttura della webapp Mooncake

La struttura del progetto deriva da quella di NPM con alcune modifiche:



4.4.2 Transazioni

Le transazioni vengono tutte effettuate lato client. Ciò è reso possibile dall'utilizzo della libreria *web3.js* e dall'integrazione con *Metamask*.

Grazie a *Metamask* gli utenti possono accedere alla Dapp mediante i loro account, senza esporre le loro chiavi private.

L'integrazione con *Metamask* è stata implementata mediante il seguente codice:

```
1 async function getAccount() { //Connecting to MetaMask
2     const accounts =
3         await ethereum.request({ method: 'eth_requestAccounts' });
4     account = accounts[0];
5     ethereumButton.remove();
6     showAccount.innerHTML = account;
7 }
```

In particolare questa funzione è registrata su un *Event Listener*, per tanto verrà eseguita nel momento in cui si clicca sul pulsante **Login with Metamask**.

Le transazioni previste vengono gestite mediante le funzioni **pay()** e **payWithBalance()**, entrambe associate ai rispettivi pulsanti.

```
1 function pay() {
2     //Emit a transaction to the contract
3     //Receive Payment while calling the method receivePay
4     contractRP.methods.receivePay(10, storePaymentA).
5         send({from: account, value: web3.utils.toWei("10"), gas:300000}).then(
6         (result) => {
7             alert(result);
8         }
9     ).catch((err) => {
10         alert(err);
11     });
12 }
13
14 function payWithBalance(){
15     //Emit a transaction to the contract
16     //Receive Payment while calling the method payWithBalance
17     contractRP.methods.payWithBalance(10, storePaymentA).
18         send({from: account, value: web3.utils.toWei("10")}).then(
19         (result) => {
20             alert(result);
21         }
22     ).catch((err) => {
23         alert(err);
24     });
25 }
```

Capitolo 5

Manuale Utente

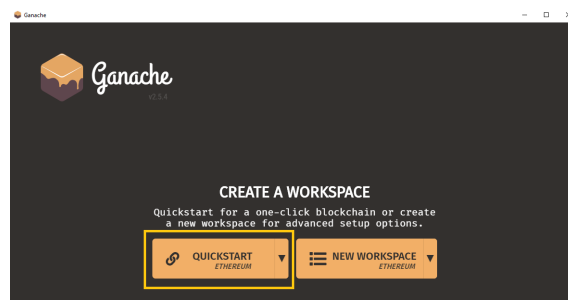
Visto l'utilizzo di tecnologie multi piattaforma, il progetto dovrebbe essere compatibile con qualsiasi sistema *Windows* e *Unix-like*, a patto che vengano rispettati i prerequisiti.

5.1 Prerequisiti

- Connessione ad *Internet*, necessaria per alcuni *CDN* presenti nel progetto, come *Bootstrap*, *JQuery* e *FontAwesome*;
- Installazione di **NodeJs** ed il suo package manager **NPM**, attraverso il sito <https://nodejs.org/en/>;
- Installazione del client **Ganache** dal sito ufficiale <https://trufflesuite.com/ganache/>;
- Installazione della suite **Truffle**.

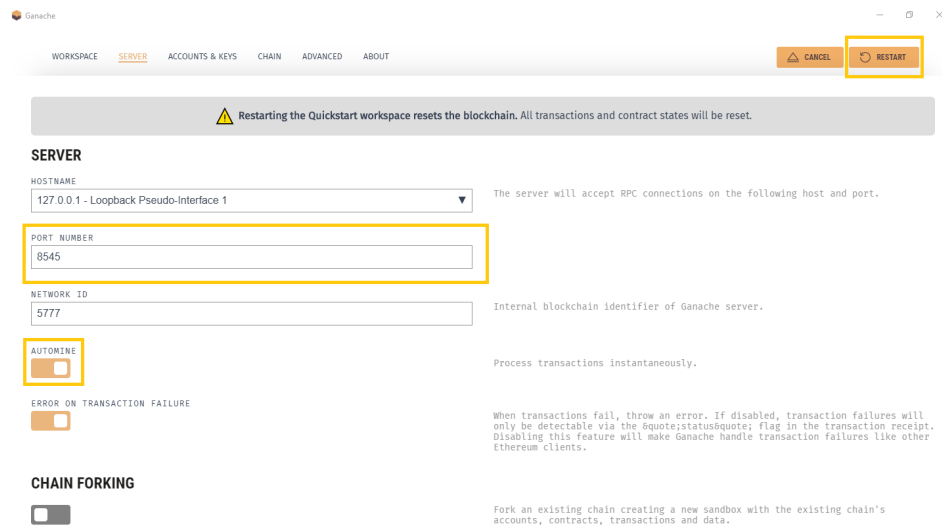
5.2 Configurare Ganache

Per prima cosa bisogna aprire *Ganache* e cliccare su "**quickstart**".



Successivamente vanno configurate alcune impostazioni del client di *Ganache*.
Aprire le impostazioni e andare nella sezione **server** per:

- settare come *numero di porta* **8545**, ovvero la porta di default per la rete di testing locale su *Metamask*;
- impostare la blockchain in **automine**, cosicché i blocchi vengano minati immediatamente;
- cliccare su **restart** per salvare le impostazioni.



Ganache

WORKSPACE **SERVER** ACCOUNTS & KEYS CHAIN ADVANCED ABOUT

Restarting the Quickstart workspace resets the blockchain. All transactions and contract states will be reset.

SERVER

HOSTNAME
127.0.0.1 - Loopback Pseudo-Interface 1

PORT NUMBER
8545

NETWORK ID
5777

AUTOMINE
☒

ERROR ON TRANSACTION FAILURE
☐

CHAIN FORKING
☐

The server will accept RPC connections on the following host and port.

Internal blockchain identifier of Ganache server.

Process transactions instantaneously.

When transactions fail, throw an error. If disabled, transaction failures will only be detectable via the `status` flag in the transaction receipt. Disabling this feature will make Ganache handle transaction failures like other Ethereum clients.

Fork an existing chain creating a new sandbox with the existing chain's accounts, contracts, transactions and data.

CANCEL RESTART

5.3 Installazione e configurazione di Mooncake

Per poter proseguire nella guida è necessario installare e configurare tutte le dipendenze di Mooncake e dunque avere il codice sorgente. Qualora non fosse disponibile basta clonare il repository da *Github*.

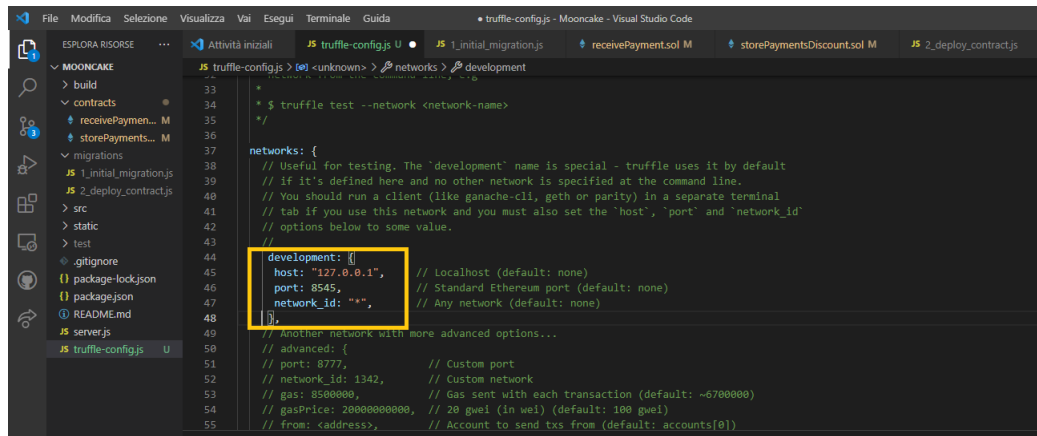
```
1 $ git clone https://github.com/jacopoesposito/Mooncake.git
```

Come prima cosa spostarsi nella cartella dove risiede il progetto e digitare il comando:

```
1 $ truffle init
```

Ciò inizializzerà l'area di lavoro.

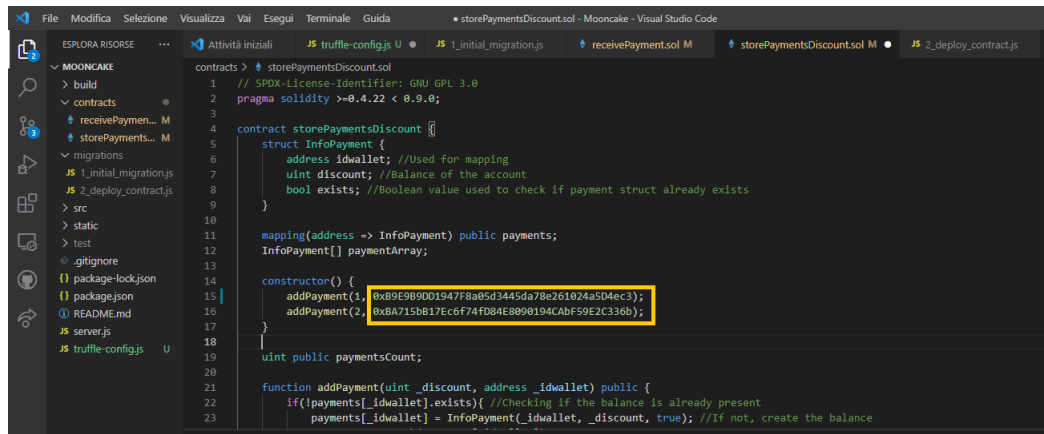
Per tutte le cartelle che truffle chiederà di sovrascrivere, negare il consenso. Successivamente spostarsi nel file **truffle-config.js** e de-commentare la parte di *development*.



```
truffle-config.js
33
34 * $ truffle test --network <network-name>
35 */
36
37 networks: {
38   // Useful for testing. The 'development' name is special - truffle uses it by default
39   // if it's defined here and no other network is specified at the command line.
40   // You should run a client (like ganache-cli, geth or parity) in a separate terminal
41   // tab if you use this network and you must also set the 'host', 'port' and 'network_id'
42   // options below to some value.
43   //
44   development: {
45     host: "127.0.0.1", // localhost (default: none)
46     port: 8545, // Standard Ethereum port (default: none)
47     network_id: "*", // Any network (default: none)
48   },
49   // Another network with more advanced options...
50   // advanced: {
51   //   port: 8777, // Custom port
52   //   network_id: 1342, // Custom network
53   //   gas: 8500000, // Gas sent with each transaction (default: ~6700000)
54   //   gasPrice: 20000000000, // 20 gwei (in wei) (default: 100 gwei)
55   //   from: <address>, // Account to send txs from (default: accounts[0])
56 }
```

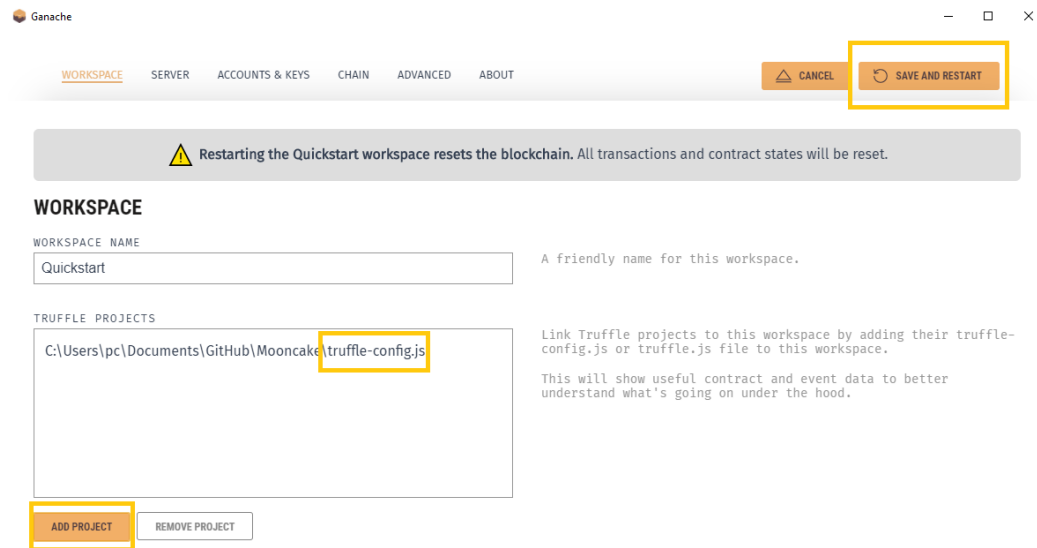
Aprire il file **contracts/storePaymentsDiscount.sol** e sostituire i due *indirizzi pubblici*, contenuti nella chiamata **addPaymentDiscount()** all'interno del costruttore di **storePaymentsDiscount()**, con quelli di due account presenti in *Ganache*.

Così fin dal primo avvio dell'applicazione si potranno testare entrambe le transazioni previste.



```
contracts > storePaymentsDiscount.sol
1 // SPDX-License-Identifier: GNU GPL 3.0
2 pragma solidity >=0.4.22 < 0.9.0;
3
4 contract storePaymentsDiscount {
5   struct InfoPayment {
6     address idwallet; //Used for mapping
7     uint discount; //Balance of the account
8     bool exists; //Boolean value used to check if payment struct already exists
9   }
10
11   mapping(address => InfoPayment) public payments;
12   InfoPayment[] paymentArray;
13
14   constructor() {
15     addPayment(1, 0xB9E9B9D01947F8a85d3445da78e261024a5D4ec3);
16     addPayment(2, 0xBA715b617Ec6f74FD84E8098194CABF59E2C336b);
17   }
18
19   uint public paymentsCount;
20
21   function addPayment(uint _discount, address _idwallet) public {
22     if(!payments[_idwallet].exists){ //Checking if the balance is already present
23       payments[_idwallet] = InfoPayment(_idwallet, _discount, true); //If not, create the balance
24     }
25   }
26 }
```

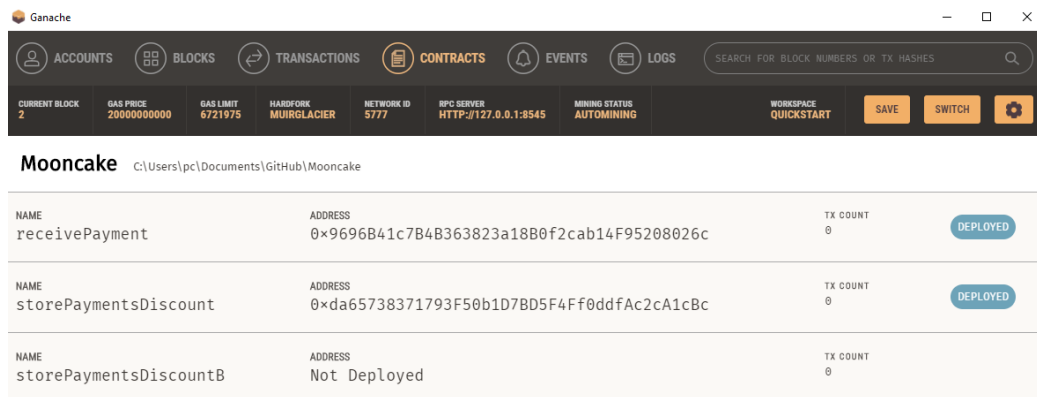
Infine spostarsi su *Ganache* e collegare all'interno delle impostazioni il progetto truffle alla nostra blockchain Ganache, così da avere conferma visiva del deploy dei contratti.



Eseguire il comando:

```
1 $ truffle migrate --reset
```

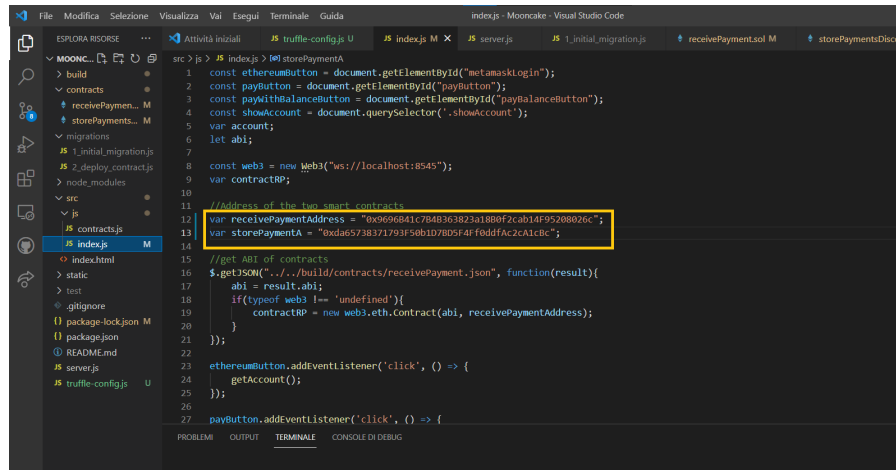
I due smart contract saranno prima compilati e poi "deployati" sulla blockchain locale di Ganache.



Adesso per installare tutte le dipendenze è necessario entrare da terminale nella cartella di Mooncake e installare i moduli presenti nel file **package-lock.json**.

1 \$ npm install

Terminata l'installazione di tutte le componenti necessarie al funzionamento della webapp, aprire il file `src/js/index.js`. Modificare le variabili contenenti gli indirizzi dei due contratti, `receivePaymentAddress` e `storePaymentA`, inserendo gli indirizzi attuali disponibili su Ganache nella sezione `contratti`.



```
1 const ethereumButton = document.getElementById("metamaskLogin");
2 const paybutton = document.getElementById("paybutton");
3 const payInitialBalanceButton = document.getElementById("payBalanceButton");
4 const showAccount = document.querySelector('.showAccount');
5 var account;
6 let abi;
7
8 const web3 = new Web3("ws://localhost:8545");
9 var contractTRP;
10
11 //Address of the two smart contracts
12 var receivePaymentAddress = "0x9696841c7b4b36382a18b0f2cab14f9520802ec";
13 var storePaymentA = "0xda65738371793f50b1d78d9f4ff0ddfac2ca1c8c";
14
15 //get ABI of contracts
16 $.getJSON(".././build/contracts/receivePayment.json", function(result){
17   abi = result.abi;
18   if(typeof web3 !== 'undefined'){
19     contractTRP = new web3.eth.Contract(abi, receivePaymentAddress);
20   }
21 });
22
23 ethereumButton.addEventListener('click', () => {
24   getAccount();
25 });
26
27 paybutton.addEventListener('click', () => {
```

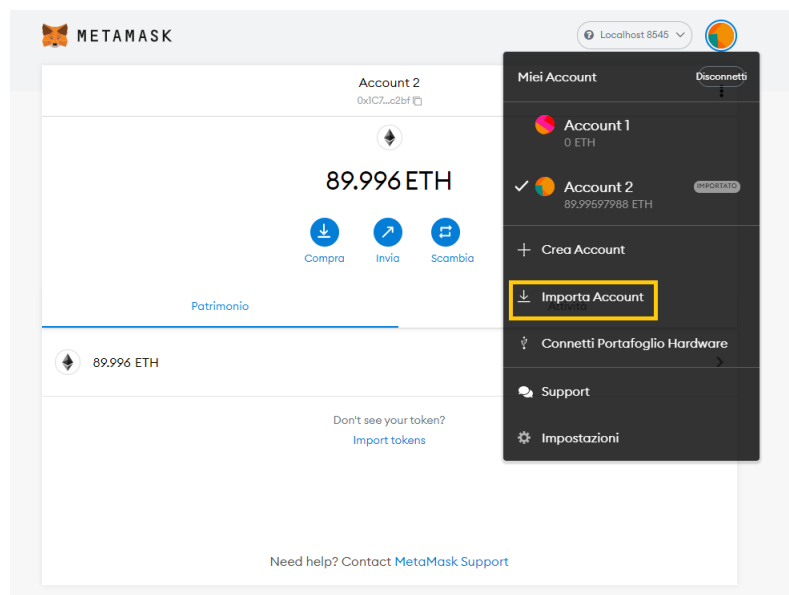
Si può ora lanciare la webapp da terminale con il comando:

```
1 $ node server.js
```

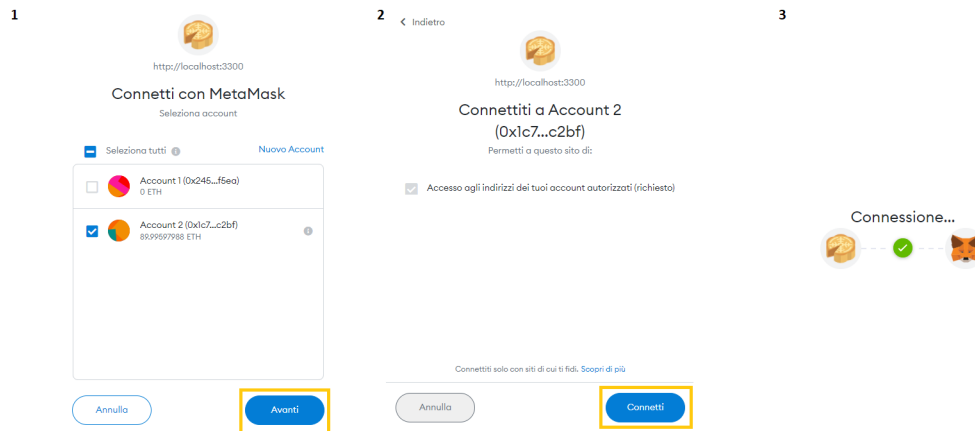
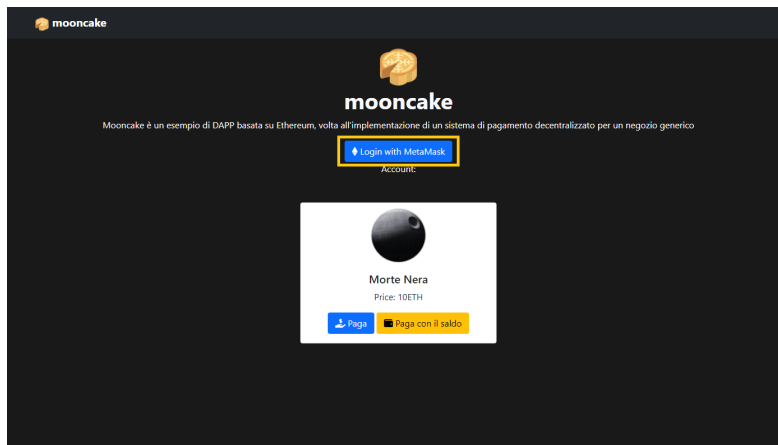
Dopo l'istallazione di *Metamask*, sarà possibile tramite browser accedere a Mooncake visitando l'indirizzo **localhost: 3300**.

5.4 Login con Metamask

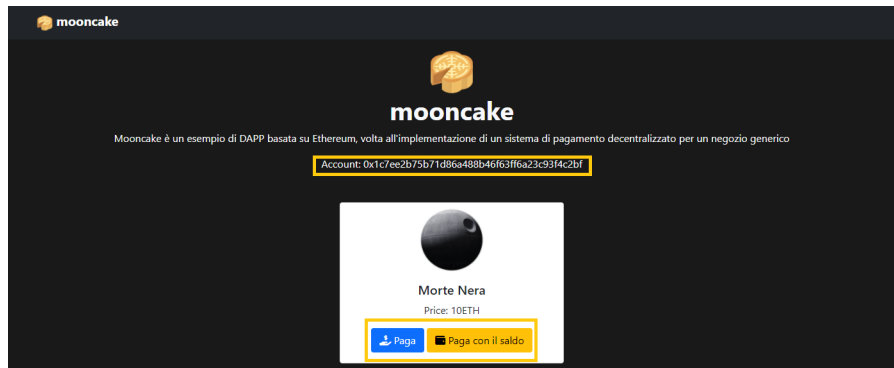
Configurare *Metamask*, importando gli account da Ganache, e assicurarsi di utilizzare la blockchain locale. Aprire l'estensione di *Metamask* e importare un account locale.



Adesso è possibile accedere alla Dapp di Mooncake collegandosi all'indirizzo **localhost: 3300**. Cliccando sul pulsante **login with Metamask** si aprirà il pop-up di *Metamask*, selezionare l'account con il quale si vuole effettuare la connessione al servizio e cliccare sul pulsante **Connetti**.



Si possono ora effettuare le due transazioni previste cliccando i pulsanti ad esse associate. Si ricorda che tutte le transazioni effettuate hanno come mittente l'account selezionato su *Metamask* in fase di connessione.



5.5 Attività sulla blockchain

È possibile avere un riscontro grafico della creazione dei blocchi e delle transazioni effettuate attraverso il client di Ganache.

Ganache - BLOCKS

CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE QUICKSTART	SAVE	SWITCH	⚙️
BLOCK 3	2000000000	8721975	MURGLACHER	5777	HTTP://127.0.0.1:8545	AUTOMINING				
BLOCK 2										
BLOCK 1										
BLOCK 0										

Ganache - TRANSACTIONS

TX HASH	FROM ADDRESS	TO CONTRACT ADDRESS	GAS USED	VALUE	ACTION
0x4fe6683d7007771c682ece07afd298ef19834bb31aeb1bdd4f753e788559b811	0x1c7ee2b75b71d86a488b46f63f6a23c93f4c2bf	receivePayment	52262	10000000000000000000	CONTRACT CALL
0xa5aeddfe61157b06d90be978b275d3e4a560e622b3bcd2dbd39e92b2186d71	0x1c7ee2b75b71d86a488b46f63f6a23c93f4c2bf	receivePayment	47979	10000000000000000000	CONTRACT CALL
0x5198788a40ca98bb3fd07ef31315c379463674a2c0936565e7067aaeba5b0a4d	0x1c7ee2b75b71d86a488b46f63f6a23c93f4c2bf	receivePayment	291500	10000000000000000000	CONTRACT CALL
0xf0c59d2b58b73c9237189b2afc71d390df2af77c73beeb3d99f956e3bf0483c5	0x3a278c030565d03fe30fe59a2a256af21e6956	CREATED CONTRACT ADDRESS 0xc51c45b199a2d0fe1a35c033ef3648ba1f1f601	864303	0	CONTRACT CREATION
0xe95b5258c760939ec5f46fab34ef253335b1dd87620ebfeca0e8b7ddae5b64bb			0.0611	0	CONTRACT CREATION