RELAZIONE TRUSTED_DATA_MULE

DESCRIZIONE GENERALE DEL PROGETTO

Questo progetto ha lo scopo di realizzare una struttura basata su blockchain che consente a due entità sender e receiver di comunicare.

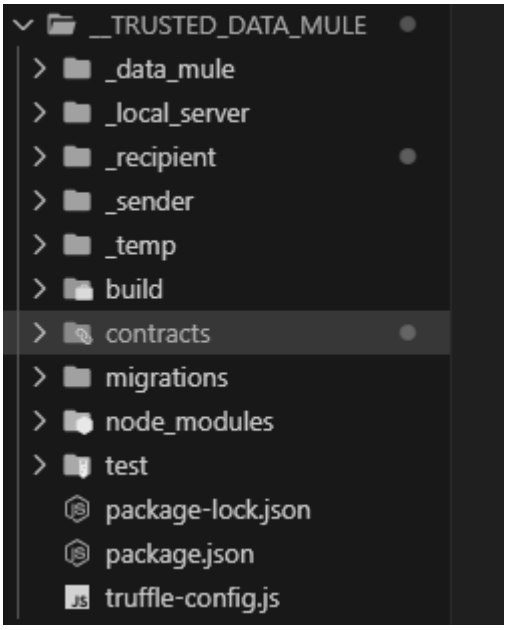
In particolare il sender non dispone di una connessione ad internet e quindi per comunicare trasferisce il messaggio che vuole inviare a un'entità chiamata Data Mule la quale si fa carico del messaggio e, quando ha una connessione ad internet, lo invia ad uno Smart Contract, quest'ultimo si farà carico di notificare il destinatario dell'avvenuta ricezione del messaggio.

Il progetto prevede anche un sistema di firma che consente allo Smart Contract di verificare che i messaggi che deve inoltrare siano effettivamente stati emessi da un entità sender registrata sullo smart contract preventivamente.

Per la comunicazione offline tra sender e data Mule è stato costruito un apposito server locale che definisce degli endpoint (richiamati dalle due entità) che consentono di leggere e scrivere i messaggi sulla cartella `_temp` del progetto.

STRUTTURA DEL PROGETTO E TECNOLOGIE UTILIZZATE

In generale come ambiente di sviluppo è stato utilizzato Visual studio code e il progetto appare come nell'immagine seguente:



Come framework per i test sulla blockchain è stata usata la truffle suite, ossia: `truffle` per scrittura, compilazione e migrazione degli smart contract; `ganache` per simulare la blockchain su cui testare gli smart contract in locale.

I singoli attori utili all asimulazione sono invece stati realizzati con:

- `_sender`: applicazione con interfaccia grafica (`react_js`)realizzata con il framework node_js e l'utilizzo delle seguenti librerie:
  - `web3` e `DataMuleContract.json`: per definire comunicazione con ganache e smart contract (in fase di registrazione dell'account) nello smart contract
  - `ethereumjs-util`: utile a generare la firma che poi dovrà essere verificata dallo SmartContract
  - `buffer`: utile a trasformare la firma da inviare in un array di byte per la trasmissione
- `_recipient`: applicazione con interfaccia grafica (react_js) realizzata con il framework node_js e l'utilizzo delle seguenti librerie:
  - `web3` e `DataMuleContract.json`: per la lettura degli eventi dallo smart contract.

Tutte le librerie (ad eccezione di `DataMuleContract.json`) sono state installate con il comando:

```
npm install <nome_libreria>
```

eseguito all’interno della cartella di riferimento.

- `_data_mule`: script node_js che esegue su terminale e comunica su console quando preleva messaggi dalla cartella `_temp` per inviarli allo smart contract.
  - `web3` e `DataMuleContract.json`: per la lettura degli eventi dallo smart contract.
- `_local_server`: server locale realizzato con il framework node_js e in particolare con le librerie:
  - `express`: per eseguire l’app come un server locale con endpoint richiamabili dall’esterno.
  - `fs` e `path`: per accedere al filesystem e scrivere nelal cartella di scambio `_temp`.
- `contracts`: cartella che contiene lo smart contract `DataMuleContract.sol` nel quale è definito il funzionamento di verifica della firma e di inoltro del messaggio. Per la verififca della firma è stato utilizzato uno smart contract preconfezionato della libreria `open-zeppelin`, in particolare:
  - `@openzeppelin/contracts/utils/cryptography/ECDSA.sol`

  installato con il comando:

  ```
  npm install @openzeppelin/contracts@4.2.0
  ```

DESCRIZIONE DEI SINGOLI ATTORI

SMART CONTRACT

Lo smart smart conntract definisce le seguenti strutture dati al suo interno:

- `senderList` (Lista dei Mittenti):

  È una lista di indirizzi Ethereum che rappresenta gli utenti registrati come mittenti. Serve per tenere traccia degli utenti autorizzati a inviare messaggi.

  ```
  address[] private senderList;
  ```

- `dataMuleTokenCount` (Conteggio dei Token dei Data Mule):

È un mapping che associa a ciascun indirizzo Ethereum dei vari DataMule il numero di token posseduti in base al numero di messaggi di cui si sono fatti carico.

```
mapping(address => uint) public dataMuleTokenCount;
```

Per il funzionamento sono invece state implementate le seguenti funzioni:

- `registerAsSender`: Permette agli utenti di registrarsi come mittenti, aggiungendo il loro indirizzo alla lista dei mittenti autorizzati.

```
// funzione che consente a un utente di
// registrarsi come sender
function registerAsSender() public {
    senderList.push(msg.sender);
}
```

- `insertAddress`: Inserisce un nuovo indirizzo nell'insieme dei contatori dei token dei Data Mule, assicurandosi che sia registrato con un conteggio iniziale di token pari a zero.

```
// Funzione per inserire un nuovo indirizzo
// in dataMuleTokenCount
function insertAddress(address _address) private {

    // Aggiunge l'indirizzo alla struttura
    // con token_count = 0
    // se non era stato inserito precedentemente
    if (dataMuleTokenCount[_address] == 0){
        dataMuleTokenCount[_address] = 0;
    }
}
```

- `incrementTokenCount`: Aumenta il conteggio dei token associati a un certo indirizzo di Data Mule quando questo invia un messaggio e la verifica della firma ha esito positivo.

```
// Funzione per asseganre un token a un certo
// indirizzo di un DataMule
```

```
function incrementTokenCount(address _address) private {

    // Si dà per scontato che l'indirizzo sia già
        // nel mapping
    // Aumenta di 1 il token_count dell'indirizzo
    dataMuleTokenCount[_address] ++;
}
```

- `verify`: Verifica la firma di un messaggio e invia il messaggio al destinatario se il mittente è autorizzato, assegnando un token al mittente come ricompensa.

```
// Verifica la firma del messaggio e invia i messaggio
// al destinatario
function verify(string memory message,
                                string memory recipient,
                                bytes memory signature)
                                    public returns (bool)

    // si riconcatenano le due parti
        // del messaggio in modo da
    // riottenere il formato in cui è
        // stato firmato il dato
    string memory concatenatedMessage
                                           = append(messa

    // Calcola l'hash dei dati da verificare
    bytes32 hash
            = keccak256(abi.encodePacked(concatenatedMessa

    // Recupera l'indirizzo dell'utente che
        // ha generato la firma
    address signer = hash.recover(signature);

    // si verifica che ci siano signer registrati
    assert(senderList.length > 0);

    // Controlla se l'indirizzo recuperato è
```

```
        // nella lista di sender
    for (uint i = 0; i < senderList.length; i++) {

        // se si trova corrispondenza
        if (signer == senderList[i]) {

            // si ricompensa il dataMule
                        // associandogli dei token
            insertAddress(msg.sender);
            incrementTokenCount(msg.sender);

            // si invia il messaggio al destinatario
                        // tramite l'evento
            emit SendMessage(message, recipient);

            return true;
        }
    }

    // si restituisce false se non si trova
        // corrispondenza tr i firmatari
    return false;

}
```

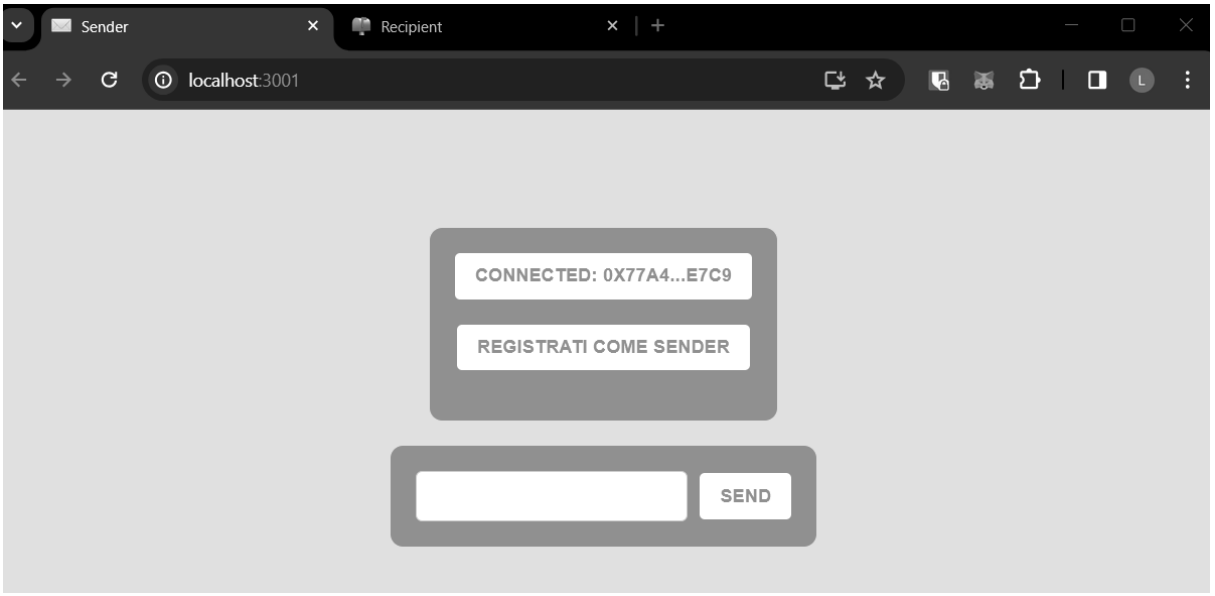
Dopo aver verificato la firma del messaggio si emette l'evento `SendMessage` grazie al quale il `_recipient` verrà notificato del messaggio ricevuto da parte dello smart contract.

- `append`: Funzione di supporto che concatena due stringhe, utilizzata per preparare il messaggio prima di firmarlo.

```
// funzione di servizio per appendere
// le due stringhe e riportarle al formato di firma
function append(string memory a, string memory b)
                                        internal pure retu
    return string(abi.encodePacked(a, "|||", b));
}
```

SENDER

L’app react appare graficamente come nell’immagine seguente:



In una fase iniziale si preme il bottone “CONNECT METAMASK ACCOUNT” per connettere un account valido; quest’ultimo che deve essere importato preventivamente in Metamask prendendo le informazioni dalla blockchain `ganache`.

Successivamente ci si registra come sender tramite bottone per aggiungersi alla lista `senderList` dello smart contract e consentire allo smart contract di verificare l’effettiva autenticità delle firme generate. La funzione utilizzata per richiamare la funzione è:

```
// funzione utilizzata per registrarsi all'interno dello smar
  async function registerAsSender(){
    const {contract} = state;

    /// si richiama la funzione per registrarsi come sender
    await contract.methods.registerAsSender().send({from: wal

    window.location.reload();
  }
```

Dopo essersi registrati come entità sender l’app consente di inviare messaggi tramite l’apposito campo di testo e il bottone “SEND”. In particolare quando si

preme il bottone viene eseguita la seguente funzione:

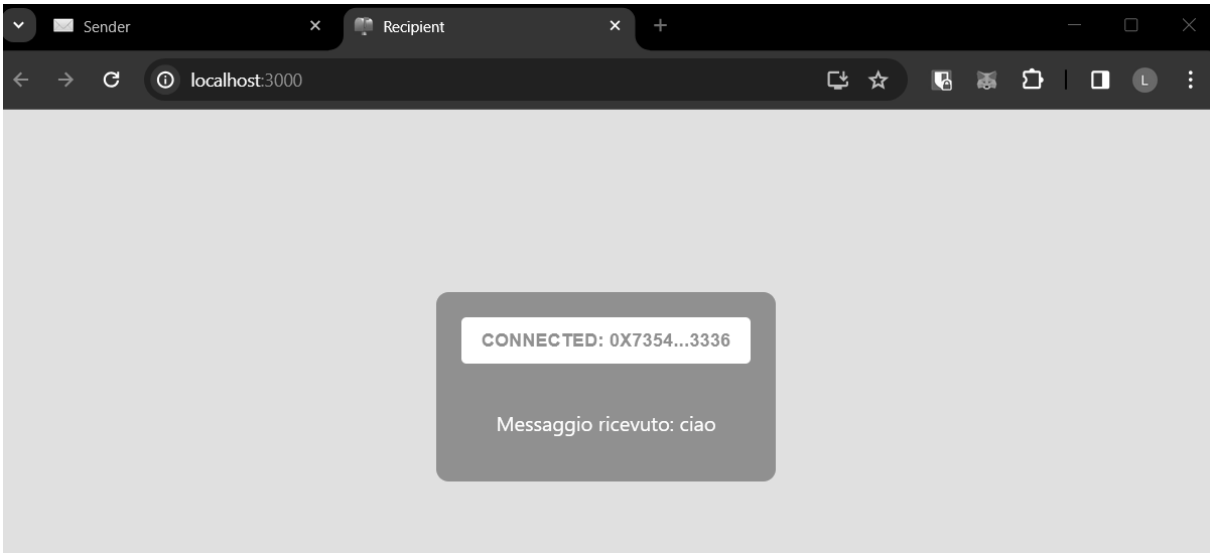
```
// Funzione utile a generare la firma del messaggio
const signAndSendInTemp = async () => {

  // si legge il valore inserito dal sender nell'interfaccia
  let text = document.getElementById('value').value;

  // si concatena al testo l'indirizzo del messaggio inserend
  text = text + "|||" + await getIndirizzoDestinatario();

  // si ottiene la chiave privata del sender
  const stringPrivateKey = await getChiavePrivataSender();
  const privateKey = Buffer.from(stringPrivateKey, 'hex');

  // Si trasforma il messaggio in un array di byte
  const bufferMsg = Buffer.from(text);

  // Calcola l'hash a partire dal messaggio
  const messageHash = ethUtil.keccak256(bufferMsg);

  // Firma l'hash del messaggio con la chiave privata
  const firma = ethUtil.ecsign(messageHash, privateKey);

  // Converti la firma in un formato accettabile dallo smart
  const signatureHex = `0x${firma.r.toString('hex')}
                        ${firma.s.toString('hex')}
                        ${firma.v.toString(16)}`;

  // si richiama il server locale per scrivere sul file
  const response_local_server
    = await fetch('http://localhost:3002/scriviMessaggioFirma
      method: 'POST',
      headers: {
          'Content-Type': 'application/json'
      },
      body: JSON.stringify({ message: text, signature: signat
  });
```

```
  // si stampa l'esito della chiamata
  if (response_local_server.ok) {
      console.log('SCRITTI FIRMA E MESSAGGIO SU FILE CON SCUC
    } else {
      console.log('!! ERRORE DURANTE SCRITTURA SU FILE  !!');
    }
}
```

La funzione dopo aver letto l'informazione dall'interfaccia di testo concatena al dato l'indirizzo del mittente, separando i due dati con il segnaposto `"|||"`. Successivamente genera la firma sul dato ottenuto e richiama la funzione l'endpoint `http://localhost:3002/scriviMessaggioFirmaSuFile` del `_local_server` per aggiungere il file con messaggio e firma all'interno della cartella `_temp`.

RECIPIENT

L'app react appare graficamente come nell'immagine seguente:



Analogamente all'app del sender, anche il recipient prevede una prima fase che consente di connettersi a un certo account della blockchain ganache tramite l'estensione Metamask del browser.

Infine ogni volta che arriva un messaggio (lo smart contract) viene stampato a schermo se l'indirizzo del destinatario corrisponde a quello connesso con l'account Metamask.

SERVER LOCALE

Come spiegato in precedenza questo server consente a `_sender` e `_data_mule` di scrivere e leggere dalla cartella `_temp` tramite due endpoint definiti come di seguito:

```
// Endpoint richiamato dal sender per scrivere i dati nel
// file che verrà prelevato dal data mule
app.post('/scriviMessaggioFirmaSuFile', (req, res) => {

  // si crea la cartelal se non esiste
  const dir = '../_temp';
  if (!fs.existsSync(dir)){
      fs.mkdirSync(dir);
  }

  const { message, signature } = req.body;

  // si associa al file un nome univoco con data e ora
  const fileName = new Date().toISOString().slice(0, -1).repl
  const filePath = path.join(dir, fileName);

  // si formattano i dati per scriverli su file
  const data = { message, signature };

  // si crea il file con i dati
  fs.writeFile(filePath, JSON.stringify(data), (err) => {
      if (err) {
          console.error(err);
          res.status(500).send('Errore durante la scrittura d
      } else {
          res.send('File scritto con successo');
      }
  });
});
```

```
// Endpoint richiamato dal Data Mule per leggere dal file i d
app.post('/leggiMessaggioFirmaDaFile', (req, res) => {
  const dir = '../_temp';
  if (!fs.existsSync(dir)){
```

```
        fs.mkdirSync(dir);
    }

    fs.readdir(dir, (err, files) => {
        if (err) {
            console.error(err);
            res.status(500).send('Errore durante la lettura del
        } else {
            if (files.length === 0) {
                res.status(404).send('Nessun file trovato nella
            } else {
                const filePath = path.join(dir, files[0]);
                fs.readFile(filePath, 'utf8', (err, data) => {
                    if (err) {
                        console.error(err);
                        res.status(500).send('Errore durante la
                    } else {
                        const { message, signature } = JSON.par
                        res.send({ message, signature });

                        // Elimina il file dopo averlo consegan
                        fs.unlink(filePath, (err) => {
                            if (err) {
                                console.error(err);
                            } else {
                                console.log(`File ${filePath} e
                            }
                        });

                    }
                });
            }
        }
    });
});
```

DATA MULE

Questo script ogni 5 secondi esegue una funzione che prova a prelevare un messaggio dalla cartella `_temp`, ossia la seguente:

```
async function main() {
  await getContractInstance();

  const dataMuleAddress = await getDataMuleAddress();
  console.log("Data mule address:");
  console.log(dataMuleAddress);

  // Esegui fetchDataAndVerify ogni tot secondi
  setInterval(fetchDataAndVerify, 5000);
}
```

```
// funzione che invia messaggi (se ci sono) allo smart contra
async function fetchDataAndVerify() {
  const response_local_server = await fetch('http://localhost
      method: 'POST',
      headers: {
          'Content-Type': 'application/json'
      },
      body: JSON.stringify({})
  });

  //
  if (response_local_server.ok) {
      const { message, signature } = await response_local_ser


      console.log('-- Dati letti da file: --', message);
      console.log('Messaggio:', message);
      console.log('FirmaHex:', signature);
      const signature_da_file = {
          r: Buffer.from(signature.slice(2, 66), 'hex'),
          s: Buffer.from(signature.slice(66, 130), 'hex'),
          v: parseInt(signature.slice(130, 132), 16)
      };
      console.log('Firma:', signature_da_file);
```

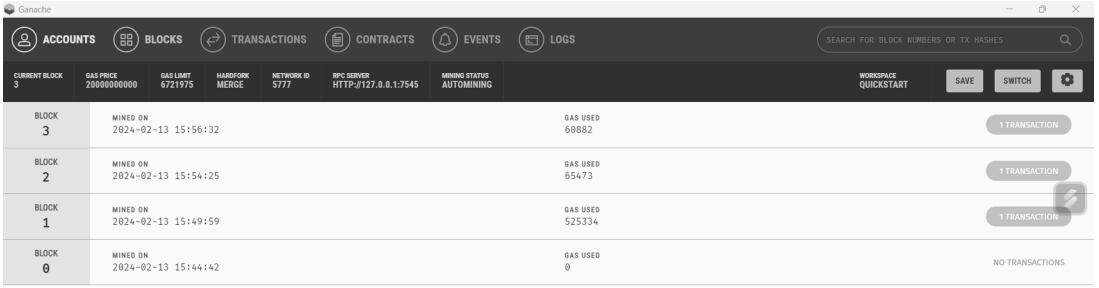
```
        // si suddivide il messaggio per inviarlo allo smart co
        let parts = message.split('|||');

        // si ottengono le due informazioni separate
        let text = parts[0];
        let addressDestinatario = parts[1];

        // si inoltra il messaggio allo SMART CONTRACT
        const dataMuleAddress = await getDataMuleAddress();
        const response_contract = await contract.methods.verify

        console.log(response_contract);

    } else {
        console.error('Nessun messaggio trovato');
    }
}
```

In particolare si richiama sempre l'endpoint `http://localhost:3002/leggiMessaggioFirmaDaFile` del `_local_ server` per controllare se ci sono dei nuovi messaggi dentro alla cartella `_temp` e si invia allo smart contract il messaggio e la relativa firma grazie alla funzione

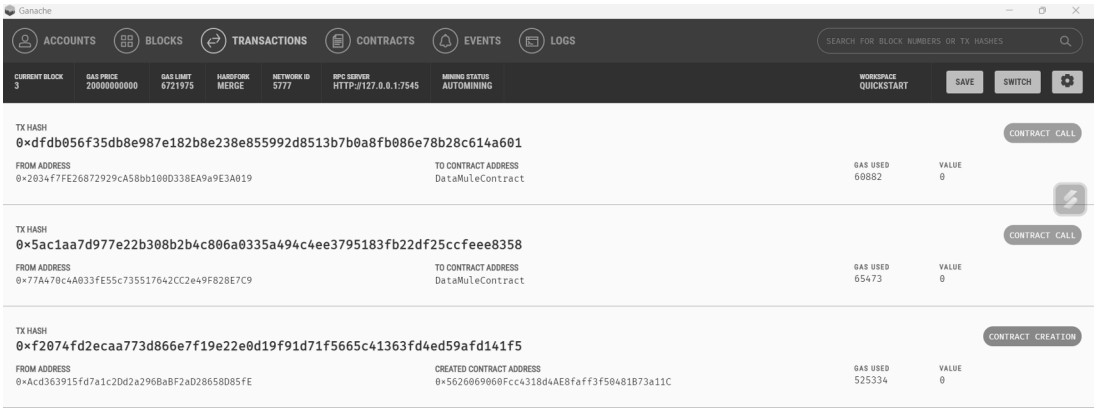
```
await contract.methods.verify(text, addressDestinatario, sign
                                                        .send
```

IMMAGINI SIMULAZIONE

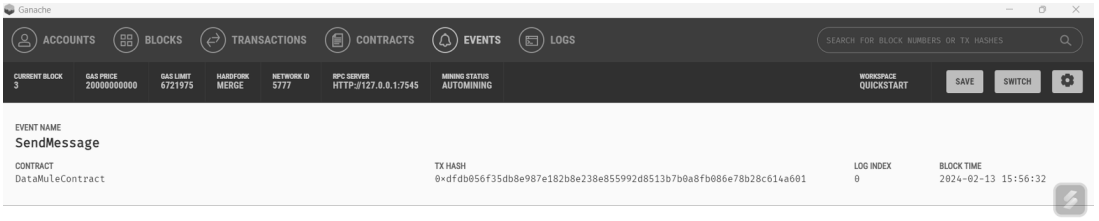
- GANACHE
    - BLOCCHI DOPO SIMULAZIONE



- TRANSAZIONI DOPO SIMULAZIONE



- EVENTI DOPO SIMULAZIONE



- TRUFFLE
  - esito comando `truffle migrate –reset`

```
> Compiling .\contracts\DataMuleContract.sol
> Artifacts written to C:\Users\lucap\OneDrive\Desktop\PROGETTI\PERZONALI\__TRUSTED_DATA_MULE\build\contracts
> Compiled successfully using:
   - solc: 0.8.0+commit.c7dfd78e.Emscripten.clang


Starting migrations...
======================
> Network name:    'development'
> Network id:      5777
> Block gas limit: 6721975 (0x6691b7)


1_DataMuleContract.js
=====================

   Replacing 'DataMuleContract'
   ----------------------------
   > transaction hash:    0xf2074fd2ecaa773d866e7f19e22e0d19f91d71f5665c41363fd4ed59afd141f5
   > Blocks: 0            Seconds: 0
   > contract address:    0x5626069060Fcc4318d4AE8faff3f50481B73a11C
   > block number:        1
   > block timestamp:     1707835799
   > account:             0xAcd363915fd7a1c2Dd2a296BaBF2aD28658D85fE
   > balance:             99.99822699775
   > gas used:            525334 (0x80416)
   > gas price:           3.375 gwei
   > value sent:          0 ETH
   > total cost:          0.00177300225 ETH

   > Saving artifacts
   -------------------------------------
   > Total cost:       0.00177300225 ETH

Summary
=======
> Total deployments:   1
> Final cost:          0.00177300225 ETH
```

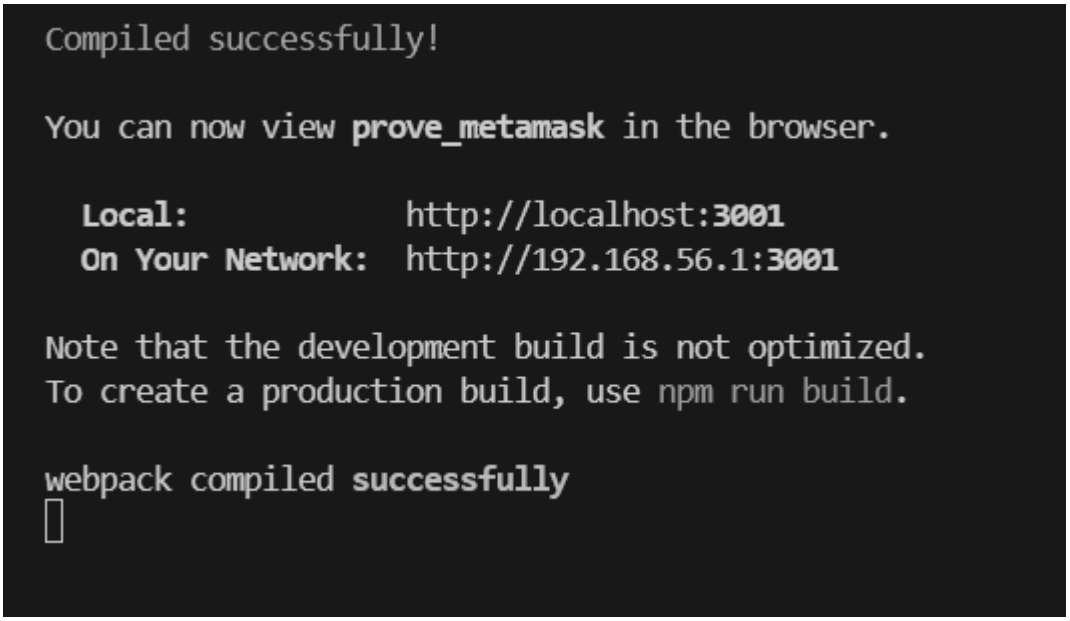
- verifica dell'asseganzione del token da `truffle console`:

```
truffle(development)> contratto.dataMuleTokenCount("0x2034f7FE26872929cA58bb100D338EA9a9E3A019");
BN { negative: 0, words: [ 1, <1 empty item> ], length: 1, red: null }
truffle(development)>
```

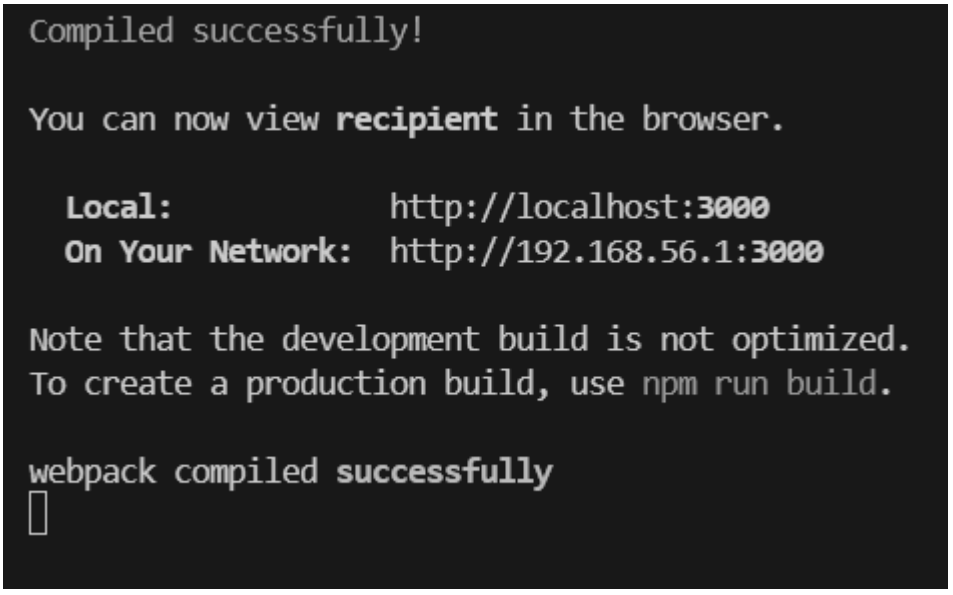
- LOCAL SERVER

```
PS C:\Users\lucap\OneDrive\Desktop\PROGETTI\PERZONALI\__TRUSTED_DATA_MULE\_local_server> node server.js
Server in ascolto sulla porta 3002
File ..\_temp\2024-02-13_14-55-26.752.txt eliminato con successo
```

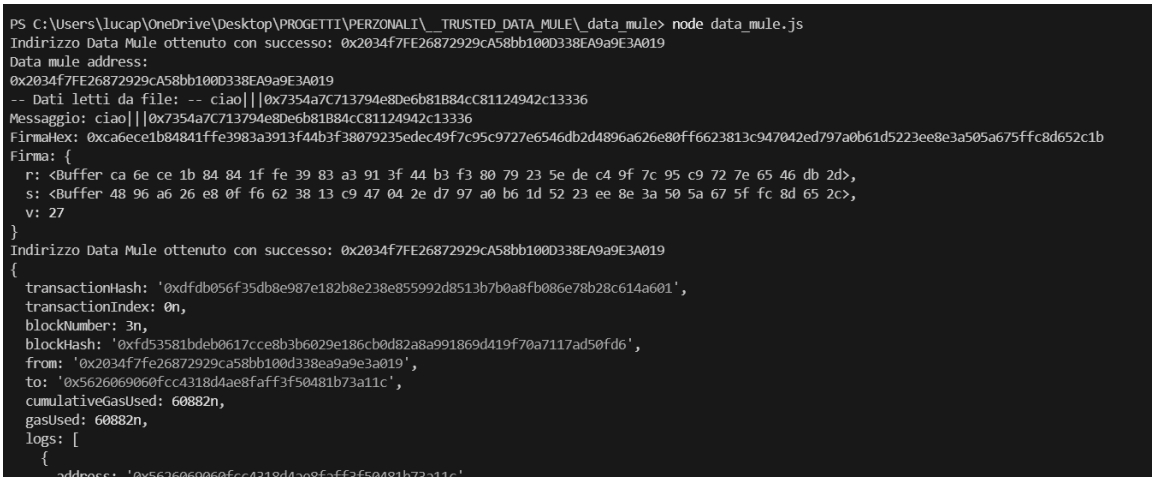
- SENDER



- RECIPIENT



- DATA MULE



```
PS C:\Users\lucap\OneDrive\Desktop\PROGETTI\PERZONALI\__TRUSTED_DATA_MULE\_data_mule> node data_mule.js
Indirizzo Data Mule ottenuto con successo: 0x2034f7FE26872929cA58bb100D338EA9a9E3A019
Data mule address:
0x2034f7FE26872929cA58bb100D338EA9a9E3A019
-- Dati letti da file: -- ciao|||0x7354a7C713794e8De6b81B84cC81124942c13336
Messaggio: ciao|||0x7354a7C713794e8De6b81B84cC81124942c13336
FirmaHex: 0xca6ece1b84841ffe3983a3913f44b3f38079235edec49f7c95c9727e6546db2d4896a626e80ff6623813c947042ed797a0b61d5223ee8e3a505a675ffc8d652c1b
Firma: {
  r: <Buffer ca 6e ce 1b 84 84 1f fe 39 83 a3 91 3f 44 b3 f3 80 79 23 5e de c4 9f 7c 95 c9 72 7e 65 46 db 2d>,
  s: <Buffer 48 96 a6 26 e8 0f f6 62 38 13 c9 47 04 2e d7 97 a0 b6 1d 52 23 ee 8e 3a 50 5a 67 5f fc 8d 65 2c>,
  v: 27
}
Indirizzo Data Mule ottenuto con successo: 0x2034f7FE26872929cA58bb100D338EA9a9E3A019
{
  transactionHash: '0xdfdb056f35db8e987e182b8e238e855992d8513b7b0a8fb086e78b28c614a601',
  transactionIndex: 0n,
  blockNumber: 3n,
  blockHash: '0xfd53581bdeb0617cce8b3b6029e186cb0d82a8a991869d419f70a7117ad50fd6',
  from: '0x2034f7fe26872929ca58bb100d338ea9a9e3a019',
  to: '0x5626069060fcc4318d4ae8faff3f50481b73a11c',
  cumulativeGasUsed: 60882n,
  gasUsed: 60882n,
  logs: [
    {
      address: '0x5626069060fcc4318d4ae8faff3f50481b73a11c'
```