**Milkchain**

E’ possibile immaginare che schedaLatte sia un oggetto con proprietà che vengono settate durante tutto l’arco della filiera.

|  |
| --- |
| **Struct** **schedaLatte** |
| Uint Id\_allevatore |
| Uint Lt\_munti |
| Uint Id\_lotto\_munto |
| Data\_mungitura |
| Flag\_malattie |
| Ph\_latte |
| Id\_responsabile\_test |
| Esito\_tests |
| Flag\_imbot\_cas |
| Id\_squadra\_imbot |
| Tipo\_latte |
| Data\_scadenza |
| Lotto\_bot |
| Lotto\_form |
| Nome\_form |
| Numero\_forma |
| Data\_caseificazione |
| Luogo\_caseificazione |
| Stagionatura |

|  |
| --- |
| **Struct Utente** |
| Id (Indirizzo\_account) |
| Mansione |
| Check\_reg |

**Fasi dello svolgimento del progetto**

**Registrazione**

1. Implementazione dello smart contract che gestisce la registrazione degli utenti.
2. Fare in modo che la blockchain ganache venga memorizzata, di modo che ogni qualvolta si esegue ganache-cli ci si interfaccia sempre con la stessa blockchain e con gli stessi accounts. Per fare ciò è necessario un comando di questo tipo:

ganache-cli -d --db "C:\Users\Utente01\Desktop\azienda\_cyber\Milkchain\blockchain\_privata"

Bisogna osservare che tramite la libreria web3 è possibile creare nuovi accounts, ma non è possibile salvarli sulla blockchain ganache. Inoltre, gli eth per ciascun account (100 eth) che vengono stampati sulla schermata iniziale coincidono con la realtà solo quando viene creata la blockchain.

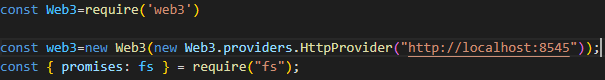
Per memorizzare le coppie di chiavi generate da ganache in un file json (keys.json) è necessario avviare ganache aggiungendo il seguente flag opzionale:

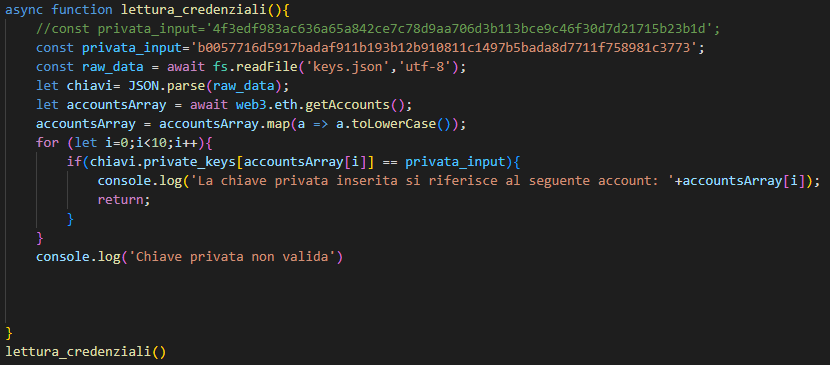
**--account\_keys\_path.json keys.json**

Tale flag permette di generare nelle directory corrente il file keys.json.

E’ utile avere questo file in quanto ci servirà per controllare il log in di un utente:

1. In fase di registrazione un utente richiede una privateKey. Il sistema deve assegnarli una tra le 10 chiavi private disponibili, a patto che tale chiave non sia già stata assegnata ad un altro utente.
2. Quando un utente registrato dovrà loggarsi inserirà la sua chiave privata nella schermata, a quel punto uno script js dovrà controllare mediante l’uso di web3 che tale chiave esista per loggare l’utente.





E’ necessario utilizzare una funzione asincrona in quanto **web3.eth.getAccounts()** è una promise. Ciò costringe ad utilizzare readFile (asincrona).

In realtà per controllare il login è necessario implementare un contratto che controlli se esiste un utente registrato che abbia la chiave privata inserita in input dall’utente.

Per richiamare un contratto all’interno di un altro contratto è necessario utilizzare la import del file .sol e utilizzare le interfacce. Per richiamare un metodo di un altro contratto è necessario passargli come parametro l’indirizzo del contratto dove è presente la funzione che bisogna richiamare.

Per fare in modo che ogni contratto acceda alle stesse interfacce è necessario creare un file solidity contenente solo le interfacce (**interfaces.sol**).

**Inserimento dati in fase di allevamento**

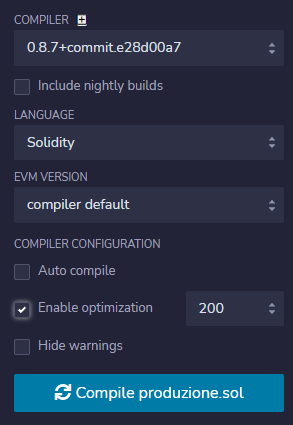
1. L’allevatore già registrato deve loggarsi nel sistema mediante la sua chiave privata. A questo livello deve agire lo smart contract che verifica la presenza dell’utente nel registro dei registrati.
2. L’allevatore inserisce i valori degli attributi di schedaLatte ( Lt\_munti, Id\_lotto\_munto, Data\_mungitura) e in modo automatico lo smart contract valorizza l’Id\_allevatore con la chiave pubblica dell’allevatore corrente. E’ necessario controllare i vari dati inseriti: per esempio i Lt siano un numero ecc. Lo smart contract creato deve accedere a dati resi pubblici sulla blockchain (per esempio la lista\_utenti) e deve poter accedere al file json dove sono memorizzate le chiavi.

**Inserimento dati in fase di produzione**

1. Il contratto produzione.sol permette di gestire le due casistiche imbottigliamento e caseificazione. Ricordiamo che nel contratto Allevamento.sol è inserito l’oggetto contenitore delle schede latte, perciò le funzioni che permettono il get e il set (o update) della scheda latte devono essere definite come **external** all’interno del contratto Allevamento.sol. La visibilità external permette al contratto produzione.sol di richiamare l’aggiornamento delle schede sia nel caso in cui il latte debba essere imbottigliato, sia nel caso in cui debba essere utilizzare per produrre una forma di formaggio. A seconda della casistica il produttore aggiornerà i dati della scheda inerenti.
2. Per semplicità ogni qualvolta è creata una nuova scheda ad essa è associata un numero **progressivo**. In questo modo un produttore può accedere ad un particolare scheda mediante il numero progressivo. In realtà si darà la possibilità di filtrare le schede anche in base all’id del lotto a cui ciascun latte è associato. Per esempio, si potranno visualizzare le bottiglie appartenenti allo stesso lotto ma aventi numero progressivo differente. In modo analogo la procedura sarà implementata per le forme

**Problematiche e risoluzioni**

Per definire la struttura delle schede latte sono necessari tanti attributi, ma solidity prevede un limite finito di variabili nella memoria. Nel caso in cui si faccia un appropriato e sovrannumerario utilizzo degli attributi il compilatore segnalerà l’errore dicendo che vi sono troppe variabili nello stack. Perciò è necessario raggruppare in struct gli attributi e definire un attributo del contratto avente una determinata struct. Inoltre, è necessario settare il compilatore in modo tale da automatizzare l’ottimizzazione delle variabili in memoria: in remix è sufficiente attivare il flag **Enable optimization**.

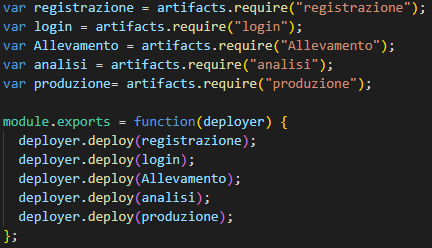


In truffle, invece, è sufficiente aggiungere l’opzione **optimizer: true**.



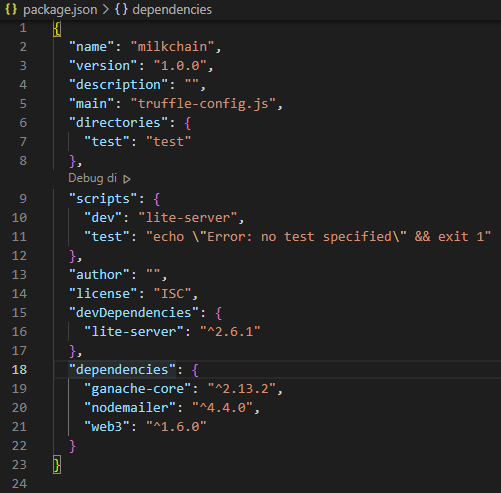
**Deploy di più contratti**

Per deployare tutti i contratti in un’unica operazione è necessario aggiornare il file 2\_deploy\_contract.js:



**Preparazione delle librerie necessarie per la costruzione della Dapp**

Nella directory dove si vuole salvare la Dapp è necessario inizializzare un progetto npm mediante il comando **npm init**. Dopodichè si devono installare le dipendenze per far interfacciare la Dapp alla blockchain ganache:



**Configurazioni necessarie per far persistere i dati sulla blockchain**

Come detto in precedenza è necessario settare le options quando si manda in esecuzione la blockchain ganache-cli. In realtà, specificando solo il percorso dove è memorizzata la blockchain locale, non è sufficiente per far persistere sulla rete anche i contratti distribuiti. Sostanzialmente, utilizzando solo le options precedentemente elencate, chiudendo la connessione alla blockchain si perde traccia dei contratti distribuiti. Dunque, sono necessarie le seguenti istruzioni:

**ganache-cli -d --db "C:\Users\Nicola Di Chiano\Desktop\azienda\_cyber\Milkchain\blockchain\_privata" -i 1234**

Ovvero è necessario specificare l’id della blockchain al quale si vuole accedere. Nel caso in cui la blockchain non è presente, si crea automaticamente una nuova blockchain situata nel path dato e avente l’id specificato.

Conseguentemente, quando si effettua il deploy dei contratti è necessario utilizzare il seguente comando che specifica il nome e l’id della rete:

**truffle migrate --network development -i 1234**

**Moduli js fondamentali per effettuare letture e scritture sulla blockchain**

Per interfacciare javascript con i contratti solidity è necessario effettuare i seguenti npm install:

* **npm i web3**
* **npm i @truffle/contract**

Di default saranno scaricate le versioni aggiornate delle librerie. Tale osservazione è fondamentale: l’attuale versione di web3 è la 1.6.0. In tal caso un oggetto istanziato mediante web3 permette l’utilizzo di **web3.eth.Contract**. Le versioni precedenti alla 1.0.0 non prevedono l’utilizzo dell’oggetto web3.eth.Contract, perciò si è riscontrato che non è possibili leggere strutture dati più complesse (come gli array) memorizzate sulla blockchain.

Seguendo la documentazione relativa a web3.eth.Contract è possibile accedere ad un contratto memorizzato mediante la seguente istruzione:

**Mycontract= new web3.eth.Contract(abi,contract\_address)**

Dopodichè l’oggetto Mycontract permette di utilizzare la proprietà methods in questo modo:

**Mycontract.methods.<nome metodo del contratto(eventuali parametri) o nome del dato pubblico sulla blockchain>**

Vediamo un esempio di lettura dell’array lista\_utenti:

ut= await mycontract.methods.lista\_utenti(0).call();

In tal caso, la funzione **call()** permette di assegnare alla variabile ut il primo elemento di lista\_utenti. Ricordiamo che in tal caso call() non effettua una transazione perciò la console di metamask non prevede l’invio di una transazione.

Per quanto riguarda la scrittura è necessario utilizzare la funzione **send():**

mycontract.methods.register(ind,mans).send({from: `${ind\_organizzatore}`})

La scrittura è una transazione perciò è necessario specificare come parametro della funzione send chi effettua la transazione.