

Relazione Progetto “Energy Market”

Giovanni Torrisi, 1000015103

Contesto

Il progetto “Energy Market” ha l’obiettivo di sviluppare una piattaforma per la gestione dell’energia elettrica in cui vari utenti possono interagire tramite smart contracts in un sistema basato su Ethereum. Per interagire con il sistema è necessario registrarsi ad esso e dopo un utente può ricoprire il ruolo di produttore registrando la quantità di energia da lui prodotta e un proprio prezzo di vendita (sia in fase di registrazione che successivamente), può ricoprire il ruolo di acquirente comprando l’energia messa in vendita da qualcun altro e può anche essere un consumatore qualora possieda dell’energia.

Tecnologie

Le tecnologie adottate sono le seguenti:

- **Solidity** : linguaggio di programmazione per lo sviluppo degli smart contracts.
- **Web3j** : è una libreria Java che permette di interagire con la blockchain Ethereum.
- **Spring Boot** : è un framework per lo sviluppo di applicazioni Java.
- **JUnit** : framework di testing per il linguaggio Java.
- **Ganache** : una blockchain locale che simula una rete Ethereum.
- **Docker** : piattaforma di containerizzazione.

Architettura di sistema

L’architettura del sistema è suddivisa in due macro-componenti: Smart Contracts e Backend in Spring Boot.

Ci sono due smart contracts, uno che gestisce lo scambio e il tracciamento dell’energia (EnergyToken), e un altro che si occupa della logica di business del mercato (EnergyMarket)

Il Backend ha un “service layer” contenente delle classi Service che incapsulano la logica di interazione con gli smart contract tramite Web3j, e delle classi runner dedicate che eseguono i deploy e altre operazioni successive (es. visualizzazione del consumo di gas a seguito di una chiamata). Inoltre, è presente anche un componente Controller che permette di interagire con l’applicazione tramite delle chiamate REST. Infine, ci sono un paio di classi di supporto che centralizzano la connessione al nodo Ethereum e il reperimento degli indirizzi.

Ganache è stato utilizzato all'interno di un container Docker.

Descrizione dei componenti

EnergyMarketApplication.java: è la classe contenente il main e rappresenta il punto di accesso al sistema.

EnergyToken.sol: Lo smart contract EnergyToken rappresenta un token ERC-20 che viene utilizzato come unità di scambio per i trasferimenti di energia. Ogni token rappresenta una quantità specifica di energia; possiamo assumere che un token equivalga ad 1 KWh. Le principali funzionalità consistono nell'emissione di nuovi token, nel loro trasferimento e nel loro consumo. Di seguito sono elencati i metodi principali:

- `function transferFrom(address from, address to, uint256 amount) external returns (bool);`
- `function mint(address to, uint256 value) public returns (bool);`
- `function burn(address from, uint256 value) public returns (bool);`

Il parametro “from” specifica l'indirizzo da cui prendere i token. Il parametro “to” specifica l'indirizzo che riceverà i token e il parametro “value” si riferisce alla quantità di token trasferita.

EnergyMarket.sol: Lo smart contract EnergyMarket è responsabile della gestione della logica di business del mercato dell'energia, astrae le operazioni di gestione dei token, si occupa della registrazione degli utenti e del prezzo di vendita. Il contratto contiene tra i suoi attributi un riferimento a EnergyToken che viene configurato all'interno del costruttore. Inoltre, è presente una struct User con campi: “pricePerToken” per specificare il prezzo di vendita dell'energia di quell'utente, e “exists” per poter verificare successivamente l'esistenza di quell'istanza. Tutte le funzioni del contratto assumono che la sorgente delle operazioni della logica di business sia l'utente che sta effettuando la transazione (msg.sender). Le funzioni principali sono le seguenti:

- **function registerUser(uint pricePerToken) external** : funzione per registrare un utente nel sistema, il parametro indica il prezzo di vendita di un'unità dell'energia di quell'utente. Inoltre, è presente un overload della funzione che non prende parametri e imposta il prezzo di default a 1 (modificabile successivamente tramite apposito metodo set)
- **function produceEnergy (uint amount) external** : funzione che un utente invoca per registrare una produzione di una certa quantità di energia, richiama il metodo “mint” di EnergyToken.
- **function buyEnergy (address producer, uint amount) external payable** : funzione che un utente invoca per acquistare una certa quantità di energia. Il parametro “producer” specifica l'indirizzo dell'utente da cui si vuole comprare. Per utilizzare il metodo è necessario inserire nella transazione una certa somma di ETH pari o superiore al prezzo di vendita del produttore moltiplicato per la quantità di energia che si vuole acquistare. Sono implementati tutti i controlli necessari per garantire che la valuta inviata sia sufficiente e qualora venga inviato più denaro di quanto necessario il metodo provvede a restituire il resto dopo aver pagato il produttore. La keyword “payable” indica che il metodo è abilitato a ricevere ETH. Richiama il metodo “transferFrom” di EnergyToken.

- **Function consumeEnergy(uint amount):** funzione per registrare un proprio consumo di energia richiama il metodo “burn” di EnergyToken.

EnergyMarket.java e **EnergyToken.java** : sono due classi wrapper degli smart contract scritti in solidity. Esse contengono delle funzioni che consentono di poter effettuare transazioni sulla blockchain direttamente dal codice Java. Queste classi sono state generate a partire dai file con estensione .abi e .bin dei contratti tramite i seguenti comandi:

```
web3j generate solidity -b src/main/resources/build/EnergyToken.bin -a
src/main/resources/build/EnergyToken.abi -o src/main/java -p com.project.EnergyMarket

web3j generate solidity -b src/main/resources/build/EnergyMarket.bin -a
src/main/resources/build/EnergyMarket.abi -o src/main/java -p com.project.EnergyMarket
```

Poiché l'utilizzo di queste classi può risultare complesso, per scelta progettuale è stato deciso di implementare delle apposite classi Service che centralizzano l'interazione con gli smart contract.

EnergyMarketService.java e **EnergyTokenService.java** : classi annotate con `@Service` che permettono di interagire con i rispettivi smart contract EnergyMarket e EnergyToken sulla blockchain, richiamando i metodi delle classi EnergyMarket.java e EnergyToken.java. Entrambe le classi hanno un attributo “contractAddress” di tipo String che contiene l'indirizzo del corrispondente smart contract sulla blockchain; grazie al fatto che stiamo lavorando su dei bean è possibile configurare questo valore anche in altre parti dell'applicazione. I metodi delle classi wrapper vengono richiamati su istanze. Per garantire un controllo sull'indirizzo mittente delle varie transazioni, ogni metodo esposto dalle classi Service ha come primo parametro una stringa “fromAddress”; la sua presenza permette di poter utilizzare la corretta istanza della classe wrapper. La chiave privata per firmare la transazione viene recuperata tramite una chiamata di metodo alla classe AccountRegistry.

Web3jConfig.java: classe annotata con `@Config` che centralizza in unico punto la creazione della connessione al nodo Ethereum. L'esistenza di questa classe permette di trattare l'istanza di Web3j come un bean e di rendere il codice più manutenibile in vista di eventuali cambi futuri del nodo.

ContractDeploymentRunner.java: classe annotata con `@Component` che effettua il deploy dei contratti. Questa classe centralizza la fase di deployment perché come già detto in precedenza lo smart contract EnergyMarket contiene un riferimento a EnergyToken ed è quindi necessario assicurarsi che venga fatto prima il deploy di quest'ultimo. Una ulteriore feature implementata consiste nel controllare se i contratti sono già stati deployati in precedenza e nell'eventualità limitarsi a farne il load, altrimenti si effettua il deploy e si conservano gli indirizzi in dei file di testo localizzati nella cartella resources/contracts. Il flusso di esecuzione consiste in due chiamate al metodo `getDeployedContractAddress()` che prende come parametro il path del file di memorizzazione, se tale chiamata restituisce l'indirizzo allora significa che era già stato fatto un deploy in precedenza, altrimenti si provvede ad effettuarlo e a fare una chiamata al metodo `saveContractAddress()`. La classe contiene dei riferimenti ai service (che ricordiamo sono gestiti come bean) e una volta che l'indirizzo degli smart contract è disponibile, esso viene settato nel corrispondente campo dei service. Il componente è annotato anche con `@Order(1)` per essere certi che venga eseguito prima di altri runner.

AccountRegistry.java: classe con la responsabilità di conservare coppie (Indirizzo, Chiave Privata). Nelle classi Service per poter fare il load dello smart contract è necessario conoscere la chiave privata

associata ad un certo indirizzo, di conseguenza è stata creato questo componente per tenere traccia di queste informazioni. Nel contesto di questo progetto, avendo utilizzato Ganache come blockchain per simulare una rete Ethereum, sappiamo che esistono soltanto dieci account e che questi sono generati dal sistema di Ganache in modo casuale sulla base di un mnemonic seed, quindi nel costruttore della classe AccountRegistry si riempie la mappa dei dati generandoli da zero a partire dal derivation path di Ganache (visibile nei log del sistema al momento dell'avvio) e da uno specifico mnemonic seed (una sequenza di 12 parole separate da uno spazio). Chiaramente per fare in modo che questo meccanismo funzioni e che ci sia coerenza è necessario creare il container Ganache specificando il mnemonic seed scelto. Se usassimo un altro ambiente rispetto a Ganache le informazioni sugli account conosciuti potrebbero essere conservate in un database o essere note utilizzando altri meccanismi. La classe è stata implementata come un Singleton.

ContractOperationRunner.java : classe runner che esegue delle operazioni base come produzione, vendita e consumo con l'obiettivo di stampare sullo schermo il consumo in gas di queste transazioni. Per farlo basta utilizzare il metodo di libreria getGasUsed() sull'oggetto di tipo TransactionReceipt restituito dalla transazione. Per garantire che questo componente venga eseguito solo dopo l'esecuzione del run di ContractDeployerRunner si utilizza l'annotazione @Order(2).

EnergyTest.java: classe con la responsabilità di eseguire i test dell'applicazione. Il file è memorizzato nell'apposita cartella in cui si posizionano i test di un progetto Spring Boot. La maggior parte dei test eseguiti sono test di integrazione in cui si verifica che le diverse funzionalità dell'applicazione funzionino bene insieme. I test includono verifiche sul corretto aggiornamento dei valori dei balance a seguito di operazioni di produzione, acquisto e consumo; e test in cui si controlla che venga effettivamente lanciata una eccezione nel momento in cui si effettuano delle operazioni non consentite (es. acquistare più di quanto un produttore possiede, interagire col sistema se non si è registrati, ecc.). Per operare sempre in un ambiente coerente è stato adottato un approccio in cui prima di ogni test viene eseguito il deploy degli smart contract, in questo modo ogni test parte da uno stato iniziale privo di interferenze derivanti da esecuzioni precedenti. Questo è stato possibile implementando un metodo di setup e utilizzando l'annotazione @BeforeEach.

Organizzazione delle directory

L'organizzazione delle directory segue tendenzialmente quella standard di un progetto Spring Boot. Prendendo come riferimento la directory root del progetto sono state aggiunte le seguenti cartelle:

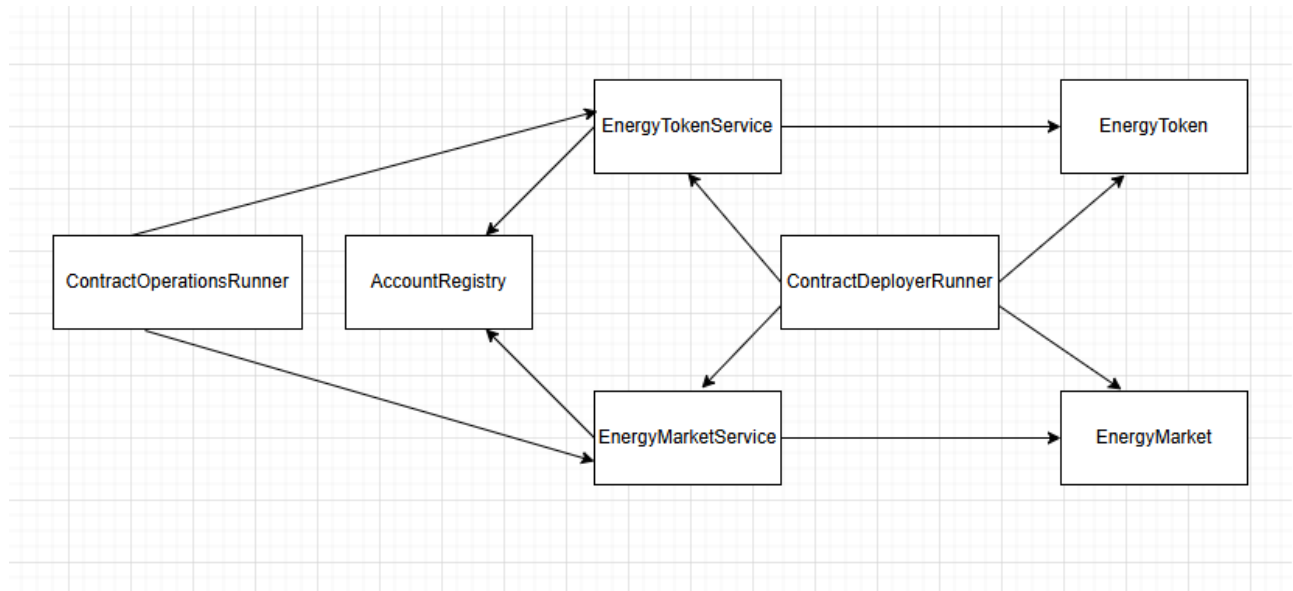
`./src/main/solidity` : contiene i file sorgenti degli smart contract

`./src/main/resources/build` : contiene i file con estensione .abi e .bin degli smart contract

`./src/main/resources/contracts` : contiene due file di testo contenenti gli indirizzi dei contratti deployati

Diagramma delle Interazioni

Di seguito è riportato un diagramma che raffigura le interazioni tra i componenti principali dell'applicazione:



Come possiamo vedere entrambe le classi Service hanno un riferimento ad AccountRegistry perché lo utilizzano per reperire le chiavi private degli account, e dato che ogni Service richiama i metodi della corrispondente classe wrapper è presente una freccia verso di essa. La classe ContractDeployerRunner mantiene riferimenti ai service, di cui aggiorna l'attributo "contractAddress" una volta che è stato fatto il deploy o che si è recuperato l'indirizzo del contratto precedentemente deployato e utilizza istanze di EnergyMarket e EnergyToken per effettuare per l'appunto le operazioni di load e deploy. Infine, la classe ContractOperationsRunner contiene dei riferimenti ai Service ed utilizza i loro metodi per effettuare le varie operazioni di interazione con il sistema.

Conclusioni e Sviluppi Futuri

L'applicazione è un sistema decentralizzato che implementa un mercato energetico in cui la valuta di pagamento sono gli ETH, e si possono svolgere operazioni di dominio come la registrazione, produzione, acquisto e consumo. Eventuali sviluppi per questo progetto potrebbero includere l'implementazione di una interfaccia grafica per interagire con l'applicazione, la gestione di più tipi di energia e l'aggiunta di ulteriore logica di business legata al mercato.