



Università degli Studi di Salerno

Dipartimento di Informatica

Tesi di Laurea di I livello in

Informatica

**Utilizzo di React per la realizzazione di
applicazioni DeFi e interazione con
smart-contract in
Solidity**

Relatore

Prof. Gerardo Iovane

Candidato

Alessandro Cavaliere

Anno Accademico 2021/2022

Abstract

A distanza di anni dalla sua nascita la Blockchain rappresenta un paradigma e una piattaforma di innovazione che permette di dare nuove risposte a tanti e diversi bisogni di imprese, organizzazioni, cittadini e consumatori. Tra questi spicca il settore della finanza decentralizzata (DeFi), con il quale è possibile implementare piattaforme che consentono forme di scambio del valore, sotto forma di token, che non richiedono garanti o intermediari. Il presente studio mira alla realizzazione di applicazioni Blockchain per la finanza decentralizzata, tramite l'utilizzo di tecnologie moderne come React e Solidity. Tali implementazioni hanno lo scopo di dimostrare che le applicazioni DeFi possono essere di pubblico utilizzo, senza la necessità di grandi conoscenze preliminari.

Le applicazioni decentralizzate (Dapp) sviluppate nel seguente lavoro di tesi sono due: la prima consiste in una piattaforma di staking di criptovalute (solo la parte front-end in React), la seconda, invece, riguarda il mondo degli exchange decentralizzati, per la precisione, degli Automatic Market Maker (AMM); la logica di quest'ultima è basata sugli smart-contract scritti in Solidity con un front-end sviluppato in React.

Indice

Introduzione	1
1 La tecnologia Blockchain	2
1.1 Che cosa si intende per Blockchain?	2
1.2 Come funziona la tecnologia Blockchain?	3
1.2.1 La struttura del blocco	4
1.2.2 L'hashing per il collegamento dei blocchi	5
1.2.3 Transazioni:il meccanismo della Digital Signatures Chain	8
1.2.4 I Merkle Trees Root e l'integrità dei dati	9
1.2.5 Il mining per la validazione dei blocchi	10
1.3 Evoluzione della Blockchain	11
1.3.1 Blockchain 1.0 : Bitcoin come moneta elettronica peer-to-peer	11
1.3.2 Blockchain 2.0 : Ethereum e la programmabilità della Blockchain tramite smart-contract (DeFi e Dapp) . .	13
1.3.3 Blockchain 3.0 : Blockchain applicata all'industria 4.0	16
2 Strumenti di sviluppo	18
2.1 Organizzazione dell'applicativo	18
2.2 Tecnologie per il back-end	18
2.2.1 Node.js	19
2.2.2 Solidity	19
2.2.3 Truffle framework	20
2.3 Tecnologie per il Front-end	21
2.3.1 React	21
2.3.2 La libreria Web3.js	22
2.3.3 Metamask	23
3 Implementazione degli applicativi	24
3.1 Introduzione e caratteristiche	24
3.2 Implementazione dello staking	24
3.2.1 In cosa consiste lo staking?	24

3.2.2	Interfaccia Grafica dell'applicativo	25
3.3	Implementazione dell'exchange	31
3.3.1	Exchange Decentralizzato (DEX)	32
3.3.2	Automatic Market Maker (AMM)	33
3.3.3	Interfaccia Grafica dell'applicativo	36
3.3.4	Interazione con gli smart-contract	48
4	Conclusioni e sviluppi futuri	59
	Bibliografia	59
	Elenco delle figure	61
	Elenco dei codici	63

Introduzione

Quando si parla di valute digitali, la prima *key word* che sorge alla mente è “Bitcoin”. Bitcoin portò alla ribalta mondiale la tecnologia Blockchain con il suoi clamorosi aumenti di valore nel corso della storia, innescando la rivoluzione degli scambi *peer-to-peer* monetari. Questi scambi monetari rappresentano l’ondata evolutiva che, negli ultimi anni, la Blockchain sta attraversando. Un’altra svolta che caratterizza il mondo della Blockchain è stata quella dell’introduzione degli smart-contract su Ethereum con il conseguente scoppio delle applicazioni basate su Blockchain, le cosiddette applicazioni decentralizzate.

L’internet, così come lo conosciamo oggi, ossia come puro mezzo di diffusione d’informazioni (Intenet of Information), si sta evolvendo in un mezzo di diffusione di valore (Intenet of Value): DeFi o finanza decentralizzata. L’obiettivo di questo elaborato è quello di fornire una panoramica dei possibili sviluppi della finanza decentralizzata con concreti esempi di implementazione di decentralized App che concernano tutte le logiche intrinseche della DeFi.

L’elaborato è organizzato come di seguito: nel capitolo 1 verrà introdotta la tecnologia blockchain, spiegato il funzionamento, la sua storia, la sua evoluzione nel tempo ed eventuali possibili sviluppi futuri.

Nel capitolo 2 verranno descritti gli strumenti di sviluppo utilizzati andando dapprima a presentare come vengono organizzati gli applicativi in termini di sviluppo per poi passare all’introduzione delle tecnologie di back-end e front-end utilizzate.

In conclusione, il capitolo 3 consiste nell’esposizione degli applicativi sviluppati, facendo, prima, un piccolo excursus della logica su cui si basano, per poi passare al puro aspetto implementativo, mostrando i codici che danno vita alle varie interfacce e rendono le operazioni possibili.

Capitolo 1

La tecnologia Blockchain

1.1 Che cosa si intende per Blockchain?

La Blockchain è una tecnologia che si basa su due concetti dell'informatica presenti da diversi decenni: la crittografia e i sistemi distribuiti. Prima di approfondire il funzionamento della tecnologia Blockchain, introduciamo brevemente questi due concetti caratteristici:

- Citando [1]: “La *crittografia* è la branca della crittologia che tratta i metodi per rendere un messaggio non comprensibile/intelligibile a persone non autorizzate a leggerlo, garantendo così, in chiave moderna, il requisito di confidenzialità o riservatezza tipico della sicurezza informatica. [...]”;
- Citando [2]: “[...]Un *sistema distribuito* è una collezione di computer indipendenti che appare ai propri utenti come un singolo sistema coerente. [...]”. Esso consiste di una serie di computer autonomi, connessi tra loro attraverso una rete e un middleware¹ di distribuzione centrale. Il quale consente ai computer di coordinare le loro attività e di condividere le risorse di sistema.

Il genio di Satoshi Nakamoto² è stato il primo a combinare queste due nozioni, in una forma mai vista prima d'ora, il tutto definito nel whitepaper³ di Bitcoin(Da [3]: “[...] è una criptovaluta e un sistema di pagamento valutario internazionale [...]” . Descritta per la prima volta in [4]. Essa è basata

¹Insieme di programmi informatici che fungono da intermediazione tra applicazioni o componenti software.

²Pseudonimo del creatore (o dei creatori) di Bitcoin

³Documento che include uno schema di un problema che il progetto sta cercando di risolvere, la soluzione a quel problema, nonché una descrizione dettagliata del loro prodotto, della sua architettura e della sua interazione con gli utenti.

sulla tecnologia che descriveremo in questo capitolo.). Una prima definizione che possiamo dare alla tecnologia Blockchain è quella di essere una struttura dati condivisa, sicura e immutabile; il modo per cui possiede tutte queste caratteristiche verrà spiegato, in dettaglio, nel paragrafo successivo.

1.2 Come funziona la tecnologia Blockchain?

Per poter capire al meglio il funzionamento della Blockchain, è doveroso analizzare il concetto di *ledger distribuito o decentralizzato* (DL).

Un *Distributed Ledger* è un *record*⁴ condiviso e sincronizzato consensualmente tra più siti, istituzioni o aree geografiche, accessibile da più persone. Esso non ammette amministrazione centralizzata o archiviazione centralizzata dei dati, il tutto viene gestito da una rete distribuita di computer, tipicamente di tipo Peer-to-Peer⁵(P2P).

Questi DL vengono realizzati grazie alla *Decentralized Ledger Technology* (DLT), quest'ultima ha come primario obiettivo quello di realizzare un grande network formato da diversi partecipanti, ognuno dei quali è chiamato a gestire un nodo della rete. Ciascun nodo è autorizzato ad aggiornare i DL in modo indipendente ma sempre sotto il controllo consensuale degli altri nodi. Il modo con cui si raggiunge questo “accordo” consensuale e la struttura dei DL sono solo alcune delle caratteristiche che connotano una DLT rispetto ad un'altra.

L’idea della prima Blockchain (Bitcoin) è quella di realizzare un network che strutturi i DL come una catena di blocchi contenenti transazioni. Questa catena non è altro che un database generale distribuito facente parte di ogni nodo della rete che, a sua volta, implementa le caratteristiche di cui sopra descritte per una DLT.

Analizzeremo ora, nel dettaglio, il contenuto che permette il funzionamento di questa tecnologia, tenendo in considerazione che i meccanismi analizzati verteranno principalmente sulle caratteristiche di Blockchain utilizzatrici dell’ *algoritmo di consenso*⁶ *Proof-Of-Work* (PoW)⁷, come Bitcoin.

⁴Un record è un tipo di dato strutturato che comprende diversi elementi (detti campi o membri) di tipo eterogeneo

⁵È un’architettura di rete detta “paritaria” in cui i nodi fungono sia da client che da server. Questi nodi sono definiti equivalenti o paritari(peer).

⁶Da un articolo dell’accademy di Binance [5]:“Un algoritmo di consenso è un meccanismo che permette a utenti o dispositivi di coordinarsi in un contesto distribuito. Deve garantire che tutti gli agenti nel sistema possano concordare su una singola fonte di verità, anche se alcuni agenti falliscono.[...].”

⁷PoW è il padre degli algoritmi di consenso, esso viene utilizzato dalle più famose reti di criptovalute.

1.2.1 La struttura del blocco

Per iniziare, diamo una panoramica generale della struttura del singolo blocco di una Blockchain. Un blocco è suddiviso in due parti distinte: L'*header* e il *body*. L'*header* del blocco, come mostrato in figura 1.1, è composto da sei campi che hanno il compito di gestire il blocco stesso:

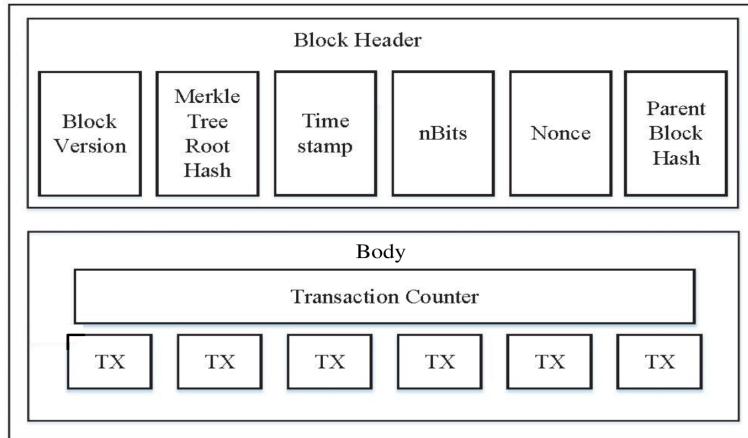


Figura 1.1: Struttura di un blocco della catena.

- La *versione*, la quale indica le regole di *validazione del blocco*⁸ da rispettare. Esse dipendono dalla versione del software che viene utilizzato per una specifica Blockchain;
- Il *Parent Block Hash*(o *Previous Hash*) è un valore di 256 bit riferito al blocco precedente della catena. Se il valore hash del blocco precedente a cui è riferito cambia, anche il PreviousHash cambierà, innescando, così, una reazione a catena di aggiornamento di identità tra tutti i blocchi successivi al blocco modificato. Verrà approfondito in (1.2.2);
- Il *Merkle Tree Root Hash* è una struttura dati con diversi use-case. Uno dei principali è quello di garantire l'integrità dei dati usufruendo dell'hashing. Verrà approfondito in (1.2.4);
- Il *timestamp*, letteralmente “marca temporale”, rappresenta un preciso istante, con lo scopo di accertare l'effettivo avvenimento di un determinato evento;

⁸L'operazione di validazione del blocco è l'elemento essenziale della Blockchain, permette l'aggiunta del blocco alla fine della catena dopo una serie di verifiche e controlli che verranno approfonditi in (1.2.5).

- Il campo *Bits*(o *difficulty target*) è la soglia *target* calcolata dall'algoritmo Proof-Of-Work di un hash di blocco valido;
- Il campo *Nonce* è un valore in byte, aggiunto al blocco e ricalcolato tramite *rehashing*⁹ in maniera tale che quest'ultimo rispetti dei criteri per la validazione del blocco. Questa operazione necessita di una grande potenza di calcolo; si tratta del famoso *mining*, approfondito in (1.2.5).

Il *body* del blocco, invece, è composto da un *contatore di transazioni* e dalle transazioni vere e proprie (indicate come “TX” nella figura 1.2.1). Queste transazioni devono essere *verificate*¹⁰. Il funzionamento e la logica delle transazioni verrà approfondito in (1.2.3).

1.2.2 L'hashing per il collegamento dei blocchi

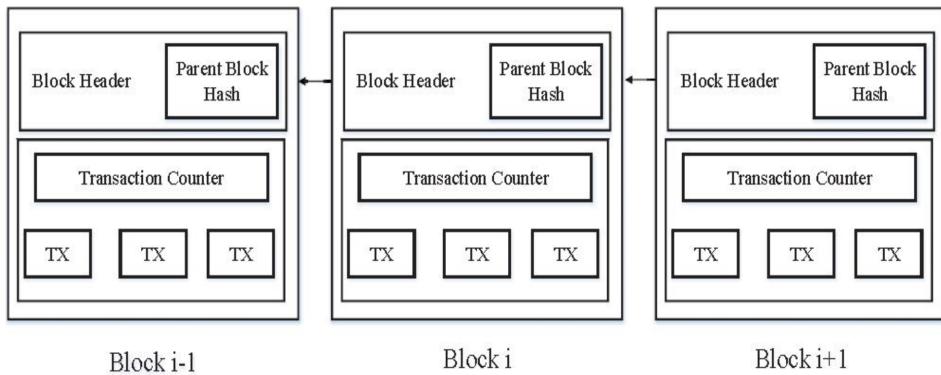


Figura 1.2: Schema base di una catena di blocchi (Blockchain).

Come accennato, la Blockchain è una struttura dati formata da una serie di blocchi collegati tra di loro(un esempio di schema base viene mostrato in figura 1.2). I nuovi blocchi si formano quando i partecipanti della rete creano nuovi dati o aggiornano quelli già esistenti. Questi blocchi memorizzano dei dati crittografati e dotati di un ID¹¹ univoco, chiamato Hash. Nella Blockchain l'hash è il collante che tiene insieme i blocchi, creando, a tutti gli effetti, una *catena di blocchi*. L'hash rende questa catena a prova di manomissioni garantendo l'integrità e l'autenticità dei dati presenti in ogni singolo

⁹Procedura che identifica delle operazioni di *hashing* ripetute nel tempo.

¹⁰La verifica di una transazione consiste nella conferma dei dettagli della transazione, inclusi il tempo della transazione, l'importo e i partecipanti.

¹¹Abbreviazione di “Identificativo”.

CAPITOLO 1. LA TECNOLOGIA BLOCKCHAIN

blocco. In questo paragrafo introdurremo il funzionamento dell'hashing in generale e l'utilizzo specifico per la tecnologia Blockchain.

Per hashing si intende il processo nel quale una qualsiasi chiave k viene inserita in una *funzione di hash* h in modo da ottenere un intero¹² $h(k)$. La coppia chiave-valore $\langle k, v \rangle$ viene memorizzata in un vettore (detto *tabella di hash*) nella posizione $h(k)$. Una funzione hash è definita come:

$$h : U \rightarrow \{0, 1, \dots, m - 1\}, \text{ dove :}$$

- h , come anticipato, è la funzione di hash;
- U è l'insieme Universo (insieme delle possibili chiavi);
- $m \in \mathbb{N}$ è la dimensione del vettore T ;
- T è la tabella di hash nel quale vengono memorizzate le coppie chiave-valore. Un esempio lo si mostra in 1.3.

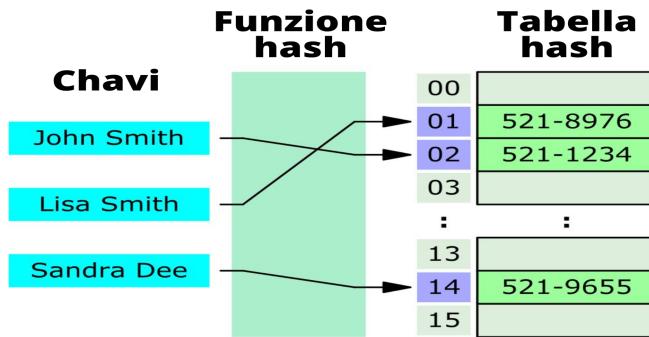


Figura 1.3: Esempio di tabella hash.

Nell'ambito Blockchain si utilizzano le *funzioni crittografiche di hash*, le quali, non sono altro che una categoria speciale delle funzioni di hash classiche. Queste particolari funzioni devono:

- essere *deterministiche* (cioè ogni messaggio possiede il sempre lo stesso hash);
- rendere univoco il valore di hash per ogni input;
- rendere molto difficile poter generare un messaggio da un dato valore di hash;

¹²In informatica, un intero viene definito come un tipo di dato utilizzato per rappresentare numeri reali che non hanno valori frazionari.

CAPITOLO 1. LA TECNOLOGIA BLOCKCHAIN

- essere veloci nel calcolare il valore di hash.

. Queste caratteristiche peculiari delle *funzioni crittografiche di hash* le rendono particolarmente adatte per garantire l'integrità, l'autenticità e la sicurezza dei dati.

Applicate al contesto Blockchain, l'hash è ciò che conferisce l'unicità al singolo blocco. Ogni blocco della catena dipende dal precedente, poiché, nel pacchetto di dati di ognuno di essi, è incluso l'hash del blocco antecedente ad esso; ciò comporta l'alterazione di tutti i blocchi successivi, ogniqualvolta, un dato blocco viene modificato dopo la registrazione iniziale dei dati per quest'ultimo. Come anticipato in (1.2.1), ogni blocco possiede un *timestamp*, che, esattamente come gli altri componenti, influenza inevitabilmente il valore hash del blocco e, come si dice in [4]: “[...] prova che i dati devono essere esistiti in quella determinata data, dal momento che sono finiti nell'hash.[...].”

Oggiorno esistono diverse *funzioni crittografiche di hash* ma col passare degli anni è stato dimostrato che alcune di esse non sono molto sicure e possiedono diverse vulnerabilità. La funzione di hash più utilizzata in ambito Blockchain è la *Secure Hash Algorithm* (SHA). Negli anni sono state segnalate collisioni e vulnerabilità per la famiglia degli SHA-0 e SHA-1, rendendoli, così, poco consigliati all'uso. Ultimi studi hanno dimostrato che le nuove applicazioni possono evitare questi problemi utilizzando le versioni più recenti degli SHA. La *funzione crittografica di hash* utilizzata per il Bitcoin è la SHA-256 (della famiglia degli SHA-2), la quale possiede un *digest*¹³ di 256 bit.

Per quanto riguarda Ethereum, prima del recentissimo aggiornamento del 15 Settembre di quest'anno al “The Merge” [6] (un piano di aggiornamento pensato diversi anni fa per migliorare la scalabilità, la sicurezza e la sostenibilità della rete Ethereum, senza compromettere la sua decentralizzazione. Si è passati al meccanismo di consenso Proof-Of-Stake (PoS)¹⁴.), la *funzione crittografica di hash* che veniva utilizzata era il Kekkak-256, esso possiede un *digest* arbitrario. Sottoinsieme della famiglia dei Kekkak è lo SHA-3 , ultima versione della famiglia di standard SHA.

¹³Un message digest è una rappresentazione numerica a dimensione fissa del contenuto di un messaggio, calcolata da una funzione hash.

¹⁴PoS è un algoritmo di consenso nel quale, ad ogni utente, viene richiesto il possesso di un certo quantitativo di criptovaluta. I *miners* vengono scelti in base alla loro ricchezza e non alla loro potenza di calcolo; questa peculiarità rende la PoS sostenibile e priva di enormi fabbisogni energetici.

1.2.3 Transazioni:il meccanismo della Digital Signatures Chain

Nella Blockchain le transazioni servono per scambiare asset di qualsiasi natura tra due o più soggetti devono essere mantenute da ogni individuo della rete e rese pubbliche tramite un messaggio broadcast. In un contesto decentralizzato, ogni transazione ha lo stesso valore d'importanza rispetto ad un'altra, di conseguenza, come è stato già anticipato in (1.2), tutti gli utenti della rete devono verificare la suddetta transazione garantendo l'autenticità della stessa. Una soluzione viene proposta da Satoshi Nakamoto in [4]: ossia quella di un sistema di *firme digitali*. Ogni nodo nella rete Peer-to-Peer (P2P) della Blockchain possiede una *chiave pubblica* e una *chiave privata*:

- La *chiave pubblica* è un valore, talvolta identificato come indirizzo, che tutti gli utenti della rete conoscono;
- La *chiave privata* è un indirizzo univoco che solo l'utente possessore ne è a conoscenza.

Per mettere in atto il sistema delle *firme digitali*, ogni transazione emessa viene firmata digitalmente utilizzando il valore hash (il modo con cui viene generato è spiegato in 1.2.2) e la chiave privata riferita al mittente. L'output generato verrà trasmesso in tutta la rete usando la chiave pubblica del destinatario della transazione. Solo il destinatario, unico possessore della chiave privata riferita alla sua chiave pubblica, potrà decifrare il messaggio criptato; così facendo è possibile anche verificare l'autenticità della transazione verificando la firma digitale generata.

Una volta che l'autenticità delle transazioni viene verificata è doveroso analizzare la maniera con cui le transazioni vengono memorizzate all'interno dei blocchi. Le transazioni vengono memorizzate all'interno di un blocco grazie al concetto di *Digital Signatures Chain* (o catena di firme digitali), il quale funzionamento, come viene espresso in [4] definisce un meccanismo per il quale :“[...]ciascun proprietario, al momento della transazione, firma digitalmente un hash della transazione precedente e la chiave pubblica del proprietario successivo[...]”. Questa procedura permette di creare una vera e propria catena di transazioni (similmente alla catena di blocchi, una struttura dati) salvata all'interno di ogni blocco della Blockchain; un'illustrazione grafica la si mostra in figura 1.4.

Come mostrato in (1.2.1), all'interno del body di ogni blocco sono presenti diverse transazioni. Queste transazioni sono incluse a loro volta in diverse catene di transazioni; queste ultime permettono di tenere traccia di tutte le operazioni finanziarie a cui sono stati sottoposti i fondi digitali di tutti i partecipanti. Come affermato nel whitepaper di Bitcoin [4] ,tutto ciò è definibile come una *moneta digitale*(o elettronica).

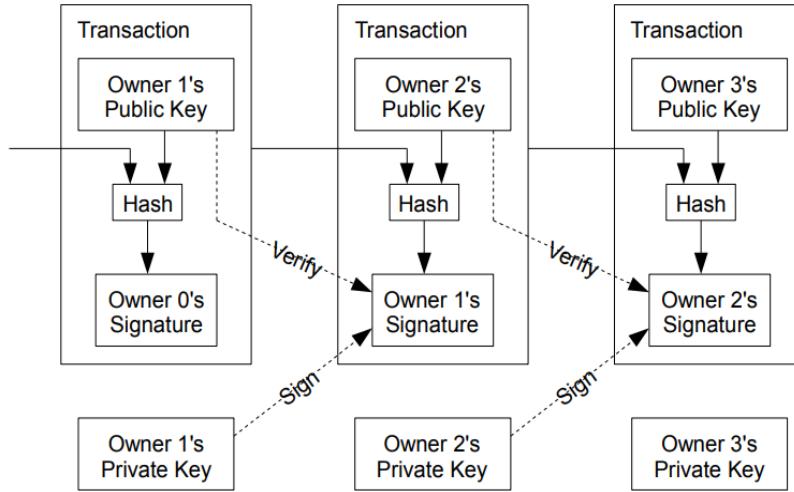


Figura 1.4: Transazioni memorizzate in un blocco tramite il meccanismo di *Digital Signatures Chain*.

1.2.4 I Merkle Trees Root e l'integrità dei dati

Un'altra componente che garantisce l'integrità dei dati presenti in un blocco è il *Merkle Tree*, il quale non è altro che una struttura dati ad *albero*¹⁵ memorizzata, sotto forma di singola radice (Merkle Tree Root), nell'*header* del blocco(come anticipato in 1.2.1). Il *bottom layer* di questa struttura ad albero , come viene espresso in un articolo scientifico sul tema [7]: “[...] mostra le transazioni memorizzate (ad es. T001 come mostrato in figura 1.5) per il blocco, che in seguito vengono convertite nelle loro firme hash SHA256 (ad es. H001 come mostrato in figura 1.5) e rappresentano le foglie dell'albero di Merkle.[...].”.

I calcoli della radice di Merkle Tree vengono effettuati attraverso il calcolo dell'hash *ricorsivamente*¹⁶ a partire dalle foglie. Una volta trovato l'hash “finale”, esso rappresenterà la radice del Merkle Tree (definita come TX_ROOT in figura 1.5).

Pertanto, se anche solo un nodo nell'albero venisse modificato, la variazione dei valori hash si diffonderebbe fino alla radice, rendendola immediatamente identificabile. La Merkle Root viene quindi utilizzata per verificare che i blocchi ricevuti dagli altri nodi siano intatti e che nessuna transazione passata sia stata alterata.

¹⁵Un albero è una struttura dati composta da nodi collegati da archi, creando una gerarchizzazione dei dati presenti al suo interno.

¹⁶In maniera ricorsiva, ossia espresso in termini di sé stesso.

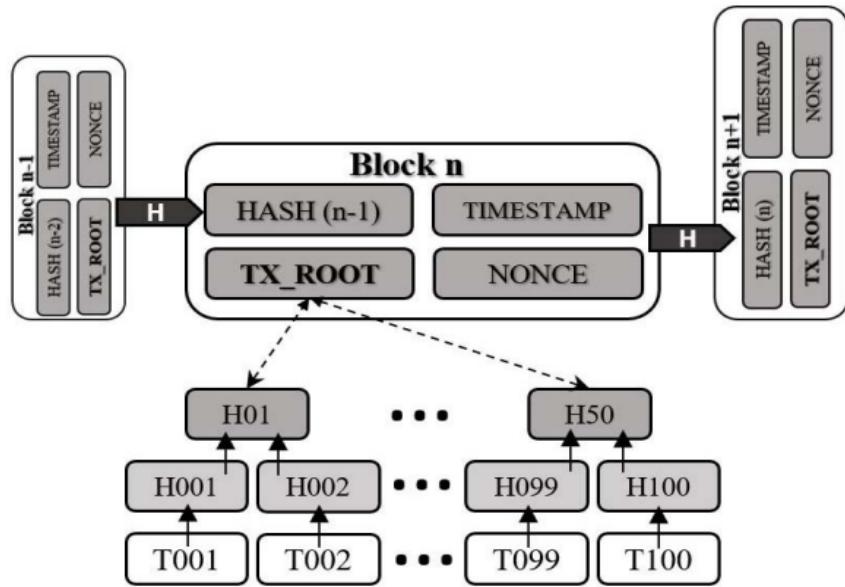


Figura 1.5: Merkle Tree Root memorizzato nell'header di un blocco.

1.2.5 Il mining per la validazione dei blocchi

Una volta capito come viene verificata l'autenticità di una transazione, è doveroso sapere come vengono *validate* una o più transazioni. Come anticipato, le transazioni devono essere condivise agli altri nodi della rete (*broadcast*) per essere verificate. Nel Bitcoin mining, esistono dei *nodi validatori* (i cosiddetti *miners*¹⁷), i quali, una volta ricevute le nuove transazioni attualmente in sospeso, ancora non validate, suggeriscono alla rete quale debba essere il nuovo blocco da pubblicare sul network. Questi *miners* per validare le transazioni e aggiungerle ad un nuovo blocco, creandolo, devono (come anticipato in 1.2.1) calcolare un valore specifico, chiamato *Nonce*, in maniera tale da soddisfare una particolare proprietà; ricordiamo, inoltre, che ogni valore presente all'interno del blocco che viene modificato, a sua volta modificherà anche il valore hash riferito al blocco stesso. Il calcolo dei miners che richiede un così grande impegno computazionale è proprio quello di generare uno specifico hash modificando il valore *Nonce* attraverso complessi problemi matematici.

Ogni blocco, dunque, insieme allo storico delle transazioni, dovrà possedere un determinato hash per essere considerato valido a tutti gli effetti (in Bitcoin, per esempio, il criterio è quello di generare un certo numero di zeri

¹⁷I miners sono proprietari di computer che contribuiscono con la loro potenza di calcolo e la loro energia alla rete di una criptovaluta basata sulla Proof-Of-Work

CAPITOLO 1. LA TECNOLOGIA BLOCKCHAIN

all'inizio del valore hash riferito al blocco). Solo i primi minatori che riescono a risolvere il problema vengono ricompensati da una quota di Bitcoin, generata proprio grazie a questo processo. Ciò significa che coloro i quali possiederanno maggiori risorse computazionali avranno maggiori probabilità di acquisire il “bottino” (block reward ¹⁸) rispetto agli altri.

Infine, il blocco viene aggiunto alla Blockchain, incrementandone la lunghezza, e le transazioni in esso contenute arrivano a destinazione.

1.3 Evoluzione della Blockchain

In questo capitolo si analizzerà la storia e l'evoluzione di questa tecnologia, si esamineranno tutte le funzionalità che, nel corso degli anni, sono state aggiunte alla Blockchain, partendo dagli albori.

Tratteremo tre diverse varianti:

- La Blockchain 1.0 : Bitcoin come moneta elettronica peer-to-peer ([1.3.1](#));
- La Blockchain 2.0 : Ethereum e l'introduzione agli smart-contract ([1.3.2](#));
- La Blockchain 3.0 : Dapp e Blockchain applicata all'industria 4.0 ([1.3.3](#));

1.3.1 Blockchain 1.0 : Bitcoin come moneta elettronica peer-to-peer

Blockchain 1.0 rappresenta la prima applicazione della tecnologia, implementata da Satoshi Nakamoto, come accennato all'inizio del capitolo 1. Questa versione è la configurazione più semplice di registro distribuito (DLT) per la registrazione di transazioni e la memorizzazione dei dati su più computer. La Blockchain di Bitcoin si limita al trasferimento di denaro tra gli utenti senza il bisogno di avere un'autorizzazione esterna (quindi senza intermediazione tra le operazioni finanziarie). Al contempo, le sue caratteristiche (che abbiamo precedentemente conosciuto) ne accentuano il ruolo di potenziale riserva di valore.

Come ha fatto Bitcoin a diventare così famoso nel mondo?

I suoi clamorosi aumenti di valori nel corso della storia e la sua politica monetaria a inflazione¹⁹ decrescente e *programmata* ha reso famoso Bitcoin

¹⁸La ricompensa data ai miners ogni qualvolta viene “minato” un blocco.

¹⁹L'inflazione è il processo in seguito al quale le valute perdono valore nel tempo, causando un aumento dei prezzi dei beni di consumo.

CAPITOLO 1. LA TECNOLOGIA BLOCKCHAIN

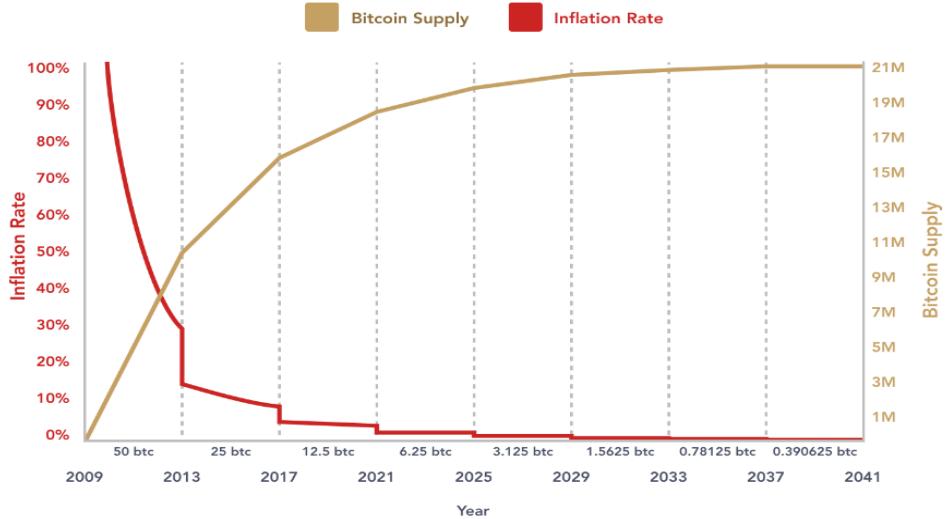


Figura 1.6: Andamento del tasso inflattivo e della *supply* di Bitcoin nel corso del tempo.

facendo parlare di sé e della sua tecnologia in tutto il mondo. Per capire al meglio come faccia il sistema ad avere una politica deflattiva, introduciamo il concetto di *halving*, ossia quella procedura che dimezza del 50% la creazione di nuovi Bitcoin ogni 210000 blocchi “minati”(circa ogni 4 anni considerando che il tempo di mining per il singolo blocco è di 10 minuti). La ricompensa iniziale, al momento della pubblicazione di Bitcoin, era di 50 BTC²⁰ per blocco, ma se questa ricompensa fosse rimasta la stessa, la valuta in circolazione sarebbe aumentata infinitamente nel corso del tempo e si sarebbe verificato, di conseguenza, una continua inflazione. Per evitare questo problema il sistema è stato *programmato* per raggiungere una *max supply*²¹ di 21 milioni di unità. Il sistema, grazie all'*halving*, quindi, rende Bitcoin più scarso col passare del tempo aumentando il suo prezzo(si evidenzia come Bitcoin sia un asset di natura deflattiva²² in figura 1.6). Di seguito si mostra il calcolo per il raggiungimento della *max supply*.

$$\sum_{i=0}^{32} 210000 \cdot \frac{50}{2^i}, \text{ dove :}$$

²⁰Abbreviazione di Bitcoin.

²¹La Maximum Supply (offerta totale) di una criptovaluta è la fornitura massima di monete che verranno mai generate.

²²Fenomeno opposto all'inflazione, quindi una diminuzione generalizzata dei prezzi, che genera un incremento del potere d'acquisto della moneta.

- la variabile i rappresenta il numero di *halving* (partendo da zero fino ai 32 *halving* stimati per il futuro);
- 210000 sono i numeri di blocchi da minare (il processo di mining viene spiegato in 1.2.5) per fare in modo che si passi all'*halving* successivo e si incrementi la variabile i ;
- il valore 50 corrisponde, come anticipato, alla ricompensa iniziale di BTC per ogni blocco minato. Quest'ultimo viene diviso per 2^i volte.

Logicamente, una ricompensa sempre più bassa comporta meno generazione di moneta e di conseguenza un maggior tempo per raggiungere la *max supply*, di fatto, la data stimata in cui verrà raggiunta si aggira intorno al 2140.

Possiamo sintetizzare la prima generazione di Blockchain come l'innesto che ha dato il via alla rivoluzione degli scambi peer-to-peer monetari. L'interesse mondiale verso Bitcoin ha posto le basi per lo studio di nuovi possibili sviluppi e use-case per questa tecnologia.

1.3.2 Blockchain 2.0 : Ethereum e la programmabilità della Blockchain tramite smart-contract (DeFi e Dapp)

La seconda generazione di Blockchain è rappresentata da Ethereum²³ [9]. Dopo il rilascio di Bitcoin, la Blockchain ha conquistato rapidamente la fantasia degli sviluppatori di tutto il mondo. Nel 2013 (circa 4 anni dopo l'ascesa di Bitcoin) un ragazzo di appena 19 anni di nome Vitalik Buterin²⁴ propone ad un team di sviluppatori un whitepaper. Dopo aver approvato la sua idea, questi sviluppatori cominciano a lavorare insieme a Vitalik per definire gli ultimi dettagli. L'anno successivo (nel 2014) viene presentato ufficialmente alla North American Bitcoin Conference a Miami il whitepaper finale di Ethereum [9], rivoluzionando la visione di questa tecnologia nel mondo.

L'idea alla base di Ethereum è quella di creare un grande sistema decentralizzato, che vada oltre il fatto di essere un semplice mezzo di pagamento o un modo per trasferire denaro, e che renda la sua struttura programmabile in toto, dall'automatizzazione degli scambi monetari a veri e propri applicativi decentralizzati.

Per rendere tutto questo possibile, sono stati introdotti gli *smart-contract* (o contratti intelligenti): programmi informatici o codici archiviati nella Block-

²³Da: [8] “Ethereum è una piattaforma decentralizzata del Web 3.0 per la creazione e pubblicazione peer-to-peer di contratti intelligenti[...]. La criptovaluta legata ad essa è Ether. Descritta per la prima volta in [9].

²⁴Sviluppatore di origini russe, ideatore di Ethereum.

CAPITOLO 1. LA TECNOLOGIA BLOCKCHAIN

chain che vengono eseguiti quando vengono soddisfatte condizioni predeterminate. In Ethereum gli *smart-contract* sono scritti in Solidity (linguaggio di programmazione approfondito al capitolo 2), uno degli strumenti cardine per lo sviluppo dell'applicazione che verrà mostrata in questo elaborato.

Il codice degli *smart-contract* viene eseguito in un ambiente chiamato *Ethereum Virtual Machine* (EVM), utilizzato da Ethereum, oltre che per rendere possibile la corretta esecuzione dei codici, anche per implementare tutte quelle funzionalità aggiuntive in maniera tale da mettere in piedi un ecosistema vero e proprio.

In una famosa pubblicazione di Apress, nella quale viene introdotto lo studio di Ethereum e della programmazione smart-contract [10], viene affermato quanto segue: “Ethereum cerca di creare un sistema in cui i modelli economici possono essere provati e dimostrati. Per il momento, Solidity sembra destinato a diventare il linguaggio de facto di tali modelli, a condizione che vengano eseguiti su una macchina virtuale globale come l’EVM.”.

Di fatto, riconosce il potenziale di Solidity e degli *smart-contract*, i quali, venendo eseguiti in ambiente Blockchain, ne ereditano tutti i vantaggi, tra cui:

- Sicurezza: La Blockchain in cui viene registrato il codice informatico del contratto intelligente non consente di essere manomesso e garantisce che possa essere eseguito in modo affidabile, generando risultati sempre verificabili e immutabili;
- Trasparenza: I record crittografati delle transazioni sono condivisi tra i partecipanti;
- Risparmio: I contratti intelligenti eliminano la necessità per gli intermediari di gestire le transazioni e, per estensione, i ritardi e le commissioni associati.

I contratti intelligenti consentono a due parti di eseguire automaticamente attività molto complesse facilitando lo scambio di valuta digitale, ereditando tutti i vantaggi della Blockchain.

In definitiva, la Blockchain 2.0 ha aperto le porte ad un'infinità di possibili implementazioni e applicazioni. Tra le più importanti ricordiamo la DeFi, argomento vertice dell'elaborato, spiegato di seguito.

Decentralized Finance (o DeFi)

La *Decentralized Finance* (o DeFi) è uno degli argomenti più discussi nelle comunità Blockchain; essa propone di disintermediare modelli finanziari tradizionali eliminando entità centrali (come banche, istituti di credito, ecc...) dando la possibilità di erogare ed usufruire di servizi finanziari in maniera

CAPITOLO 1. LA TECNOLOGIA BLOCKCHAIN

del tutto decentralizzata. Questi servizi finanziari vengono erogati dalle Decentralized Application²⁵ (o Dapp) , ossia delle web-application basate su reti P2P che sfruttano le potenzialità della Blockchain.

Come le normali applicazioni, le Dapp sono composte da un front-end²⁶, un back-end²⁷ e da un database (in questi casi è la Blockchain o gli smart-contract a fare da database). La particolarità di queste applicazioni è che, sfruttando la Blockchain, i dati inseriti dagli utenti vengono memorizzati su un registro distribuito in maniera del tutto automatica, grazie all'utilizzo degli *smart-contract* precedentemente introdotti.

Le applicazioni DeFi sono divisi il 5 livelli distinti, come definito in [11]:

- Il primo livello è il *settlement layer* (o livello dei regolamenti) in cui vengono regolate tutte le transazioni. Su questo livello si trova la Blockchain e il suo relativo asset nativo (ad esempio, Ether per la Blockchain di Ethereum). La Blockchain, come si dice in [11]:“può essere vista come la base per l'esecuzione senza fiducia e serve come livello di risoluzione delle controversie.”;
- Il secondo livello è l'*asset layer*. In questo livello sono presenti tutti gli asset emessi dal livello di regolamento, solitamente indicati come token, fungibili²⁸ e non fungibili come gli NFT(acronimo di Not-Fungible-Token ,anche chiamati “certificati digitali”, da [12] sono: “asset crittografici su una Blockchain con codici di identificazione e metadati unici che li distinguono gli uni dagli altri. Essi non possono essere scambiati in modo equivalente.”);
- Il terzo livello è il *protocol layer* (o livello dei protocolli), tale livello contiene un insieme di regole e standard concordati che normano le transazioni. Queste regole e standard sono implementati da una serie di *smart-contract*;
- Il quarto livello è l'*application layer* (o livello di applicazione), il livello nel quale si posiziona l'applicazione implementata per questo elaborato. Di fatto, l'*application layer* crea applicazioni che si collegano ai protocolli tramite l'*interazione agli smart-contract*, rendendo i protocolli molto più facili da usare;

²⁵Applicazioni che hanno il proprio codice di back-end in esecuzione su una rete peer-to-peer decentralizzata e ciò la differenzia della maggior parte delle comuni applicazioni il cui codice di back-end è in esecuzione su server centralizzati.

²⁶La parte visibile all'utente di un programma e con cui egli può interagire.

²⁷La parte non visibile all'utente di un programma che gestisce le funzioni dell'applicazione “dietro le quinte”.

²⁸Un token si dice fungibile quando può essere duplicato un numero infinito di volte in copie identiche e intercambiabili.

- Il quinto livello, infine, è l'*aggregation layer* (o livello di aggregazione) il quale fornisce strumenti per la comparazione e la valutazione dei servizi, consentendo agli utenti di svolgere attività altrimenti complesse.

Si mostra in figura 1.7 una rappresentazione grafica della struttura di un'applicazione DeFi.

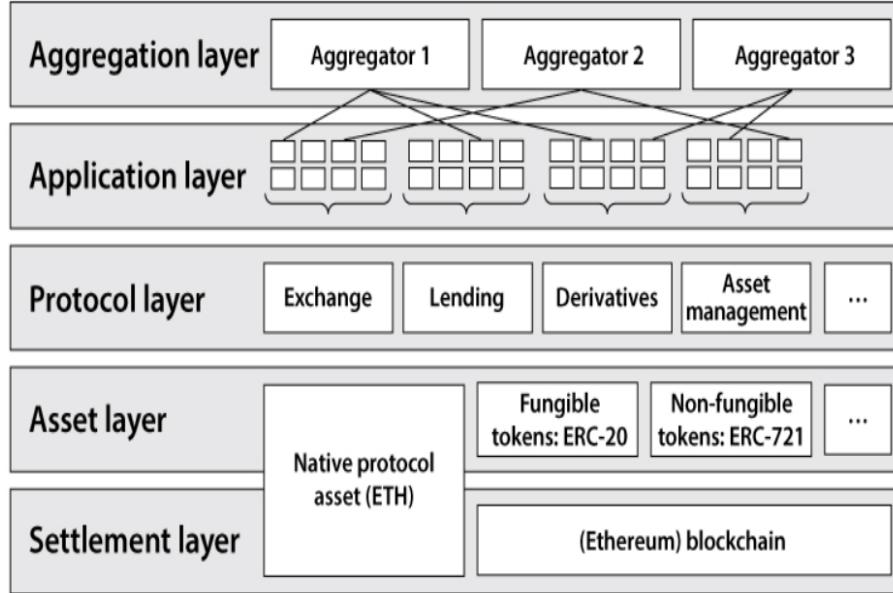


Figura 1.7: Struttura di un'applicazione DeFi.

1.3.3 Blockchain 3.0 : Blockchain applicata all'industria 4.0

Gli sviluppi successivi alla Blockchain 2.0 non hanno una definizione molto chiara, trattandosi di argomenti in continuo sviluppo. Oggi come oggi, potremmo comunque definire la Blockchain 3.0 come quell'ondata evolutiva che cerca di oltrepassare e ottimizzare i servizi delle prime due generazioni, rendendo, al contempo, la tecnologia Blockchain utilizzabile per le esigenze aziendali. Soprattutto riferiti all'industria 4.0, la quale, in breve, non è altro che, come viene detto in [13]: “la propensione dell’odierna automazione industriale ad inserire alcune nuove tecnologie produttive per migliorare le condizioni di lavoro, creare nuovi modelli di business, aumentare la produttività degli impianti e migliorare la qualità dei prodotti. [...]”. Questa particolare rivoluzione industriale richiede (come d'altronde in generale nel prossimo futuro) un grado sempre più alto di sicurezza della privacy e dei dati e maggiore fiducia. Qui entra in gioco la Blockchain.

CAPITOLO 1. LA TECNOLOGIA BLOCKCHAIN

Le principali preoccupazioni per questa generazione di Blockchain sono la sostenibilità, la scalabilità, l'economicità, una maggiore decentralizzazione e una maggiore sicurezza.

La sostenibilità, che è un argomento di forte attualità, influenza le scelte delle nuove famiglie di Blockchain. Molte Blockchain, come è stato già anticipato nei capitoli precedenti, necessitano di grandi potenze di calcolo per alcune operazioni (si pensi, per esempio, al mining) comportando un grande dispendio energetico. Non a caso le Blockchain si stanno aggiornando tenendo in considerazione questo fattore(basti guardare al Merge di Ethereum [6]), passando a modelli più sostenibili come il Proof-Of-Stake (PoS).

In conclusione, al netto di tutti questi incredibili possibili sviluppi, la tecnologia Blockchain ha ancora molto da offrire al mondo che verrà, rendendola, di fatto, una delle tecnologie più interessanti da approfondire.

Capitolo 2

Strumenti di sviluppo

2.1 Organizzazione dell'applicativo

Una volta compreso il funzionamento e le peculiarità della tecnologia Blockchain, ci indentriamo nello sviluppo vero e proprio dell'applicazione. Come anticipato, l'applicazione è una Dapp DeFi strutturata sulla rete Ethereum (per la precisione sulla rete di test Ropsten.); composta da un front-end in React e un back-end formato da smart-contract (memorizzati su Blockchain) scritti in Solidity, il quale, ci consente di distribuire l'applicazione sia su mainnet Ethereum che su qualsiasi altra Blockchain EVM compatibile. In questo capitolo verrà mostrato come le due controparti sono collegate tra di loro e tutte le tecnologie specifiche utilizzate.

2.2 Tecnologie per il back-end

In una Dapp, gli smart-contract “sostituiscono” il back-end delle applicazioni tradizionali per evitare i server centralizzati e agire in modo distribuito (P2P), invece che utilizzare il protocollo HTTP per comunicare. Gli smart-contract di Ethereum consentono di costruire architetture in cui una rete di contratti si trasmette dati tra loro, leggendo e scrivendo le proprie variabili di stato man mano che procedono, con la loro complessità limitata solo dal solo *block gas limit*²⁹. Deve essere, inoltre, considerato che uno smart-contract è impossibile da modificare dopo essere stato distribuito, può soltanto essere cancellato se è stata aggiunta in precedenza una particolare funzione (per la precisione la SELFDESTRUCT opcode³⁰).

²⁹Il limite di gas definisce il limite superiore del gas destinato al consumo di una transazione o per l'intero blocco.

³⁰un'operazione a livello della EVM, indipendente dal tipo di linguaggio o client che stai utilizzando

2.2.1 Node.js

Node.js è un ambiente di runtime JavaScript open source usato per eseguire codice JavaScript all'esterno di un browser web. Ai fini del nostro progetto, node.js ci servirà per il Node Package Manager (NPM) come pre-requisito all'installazione di Truffle (2.2.3), del React App (2.3.1) e della libreria web3.js (2.3.2).

2.2.2 Solidity

Per l'implementazione degli smart-contract su rete Ethereum si utilizza Solidity, un linguaggio di programmazione object-oriented³¹ designato specificamente per il loro sviluppo (vengono eseguiti sulla macchina virtuale di Ethereum) [14]. Questo linguaggio (che è di alto livello) è stato pensato per rendere più semplice la scrittura di smart-contract, i quali, devono essere compilati in bytecode EVM (ossia codice di basso livello interpretabile solo dalla macchina e non dall'uomo).

Per fare questo, c'è bisogno di un Contract ABI (o Application Binary Interface), ossia un'interfaccia che permette di interagire con uno smart-contract tramite uno schema standard, definendo i nomi delle funzioni e i tipi di dati degli argomenti. L'ABI, infatti, è scritto in formato JSON³².

Nella nostra Dapp, dunque, quando si vogliono inserire o leggere dei dati sulla Blockchain, le funzioni chiamate devono passare sempre per l'ABI che fa da intermediatore con l'EVM. Si mostra in figura 2.1 il processo illustrato graficamente.

Un vantaggio di Solidity è che i codici sorgente realizzati sono accessibili pubblicamente nella Blockchain di Ethereum (consultabili su qualsiasi block explorer, ossia: come detto in [15]: “è uno strumento che fornisce analisi dettagliate su una rete Blockchain[...]”, esso funge da un motore di ricerca nel quale si possono controllare i singoli blocchi, le transazioni e gli indirizzi pubblici). Per quanto riguarda la visualizzazione di questi codici: se essi sono verificati, viene mostrato il codice sorgente, altrimenti vengono mostrati in bytecode.

Solidity oltre ad essere stato uno dei primi linguaggi di programmazione smart-contract, è il più utilizzato al momento per il suo scopo e viene considerato, inoltre, il più versatile tra tutti.

³¹Letteralmente “orientato agli oggetti”. L'object orientation è un paradigma di programmazione che si basa sul concetto di classi e oggetti.

³²Acronimo di JavaScript Object Notation, è un formato standard utilizzato per lo scambio e la memorizzazione di dati sottoforma di coppie o array attributo-valore.

CAPITOLO 2. STRUMENTI DI SVILUPPO

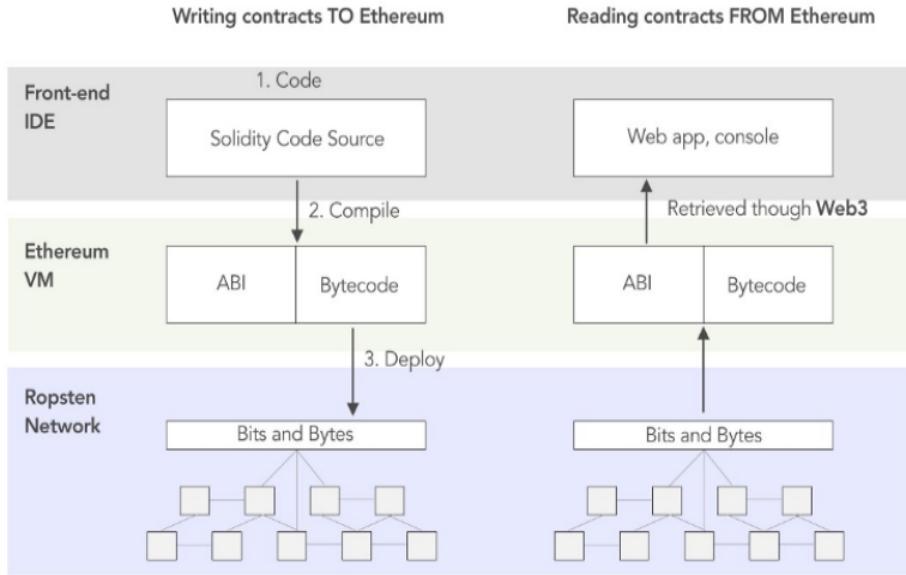


Figura 2.1: Processo di lettura/scrittura delle funzioni di uno smart-contract.

2.2.3 Truffle framework

Durante l'intero ciclo di vita del progetto, è stato utilizzato il framework Truffle della Truffle Suite di Consensys³³ [16] per il supporto allo sviluppo e alla distribuzione degli smart-contract permettendo di comunicare con essi, senza faticose programmazioni lato client. Truffle permette diverse funzioni:

- Gestire i propri contratti collegando librerie e diverse applicazioni Ethereum;
- Automatizzazione del testing dei contratti per uno sviluppo più rapido;
- Distribuzione e Migrazione degli script. La migrazione aiuta i contratti ad essere distribuiti e salvati sulla rete Ethereum, Truffle gestisce le migrazioni tramite dei file Javascript ³⁴, i quali, da documentazione [17]: “[...]sono responsabili della gestione temporanea delle attività di distribuzione e vengono scritti partendo dal presupposto che le esigenze di distribuzione cambieranno nel tempo.[...]" ;

³³Una delle principali aziende di sviluppo software Blockchain di Ethereum.

³⁴Linguaggio di programmazione multi paradigma orientato agli eventi.

```
module.exports = {
  networks: {
    ropsten: {
      provider: new HDWalletProvider(mnemonic, rpcURL),
      network_id: 3,           // Ropsten's id
      gas: 4000000,            // Ropsten has a lower block limit than mainnet
      confirmations: 2,        // # of confs to wait between deployments.
      timeoutBlocks: 200,      // # of blocks before a deployment times out
      skipDryRun: true         // Skip dry run before migrations?
    }
  }
},
```

Figura 2.2: File di configurazione *truffle-config.js* per la testnet Ropsten.

- Fornisce una console interattiva per compilare, testare o debuggare³⁵ i propri contratti.

Truffle distribuisce l'applicazione su un nodo che rappresenta la rete (Blockchain) specifica da voler utilizzare; il tutto viene definito in un unico file Javascript di configurazione chiamato *truffle-config.js*. Nel nostro caso l'applicazione è stata distribuita sulla rete di test Ropsten di Ethereum, il file di configurazione è mostrato in figura 2.2.

2.3 Tecnologie per il Front-end

A differenza del back-end, in una Dapp il funzionamento lato front-end e l'interfaccia utente (UI) non cambiano rispetto una web-app tradizionale, l'unica differenza la fa l'interazione con il back-end. La gestione delle interazioni con la rete Ethereum nell'applicazione vengono assegnate alla libreria Javascript *web3.js* e all'estensione del browser *Metamask* (spiegati nei paragrafi successivi).

2.3.1 React

La nostra applicazione è stata realizzata interamente in *React* [18], una libreria Javascript per la creazione di UI sviluppata nel 2013 da Facebook; la libreria è ora open-source ed è sostenuta da una vasta comunità di programmatore. React permette di creare interfacce dinamiche complesse che allo stesso tempo risultano essere semplici e intuitive da utilizzare.

³⁵Procedura molto importante che permette di facilitare la ricerca di errori logici all'interno del programma, viene effettuato attraverso un' altro programma chiamato debugger.

CAPITOLO 2. STRUMENTI DI SVILUPPO

Come è ormai risaputo, per la creazione di applicazioni Web è necessario coinvolgere i tre linguaggi fondamentali:

- HTML (o Hypertext Markup Language) per strutturare il sito web;
- CSS (o Cascading Style Sheets) per la resa grafica dell'applicazione;
- Javascript, invece, per la logica applicativa e per rendere interattive le applicazioni web.

Nelle applicazioni *React*, tutto questo viene semplificato nella sintassi JSX (Javascript XML), un'estensione della sintassi di Javascript, che ci permette di inserire delle strutture simili ad HTML nello stesso file in cui si scrive la logica Javascript. L'uso di JSX, seppur non obbligatorio, permette una più facile lettura degli elementi e dei suoi attributi. Questo particolare tipo di sintassi non è decifrabile nativamente dai browser web e c'è bisogno di un pre-compilatore per rendere leggibile il tutto; la libreria *React*, oggi come oggi, utilizza Babel³⁶ per questo scopo.

Una delle peculiarità di React è quella di comporre le sue applicazioni di *componenti* riutilizzabili permettendo di semplificare lo sviluppo, di non ripetere frazioni di codice più volte e di importare un determinato componente solo quando è davvero necessario. Ogni componente può essere personalizzato a proprio piacimento incorporando un CSS specifico.

2.3.2 La libreria Web3.js

Come anticipato nei paragrafi precedenti, la libreria *web3.js* [19] viene utilizzata per l'interazione con gli smart-contract su Ethereum.

Come viene espresso in [19] *web3.js* è :“una raccolta di librerie che consentono di interagire con un nodo Ethereum locale o remoto utilizzando HTTP, IPC o WebSocket.”, essa è l'API³⁷ Javascript ufficiale di Ethereum. Prima di definire nel dettaglio il funzionamento e le caratteristiche di questa libreria, introduciamo brevemente il concetto di JSON-RPC.

JSON-RPC (Remote Procedure Call Protocol) è un protocollo per la chiamata di procedure remote, nelle quali è possibile passare anche oggetti complessi e ricevere come output dati multipli. I codici di successo e di errore sono standardizzati, in maniera tale da renderne più facile la comprensione.

Questo servizio viene interrogato tramite HTTP o TCP/IP e restituisce una serializzazione del risultato in formato JSON. Ogni richiesta è caratterizzata dalle seguenti componenti:

³⁶Compiler Javascript.

³⁷Abbreviazione di Application Programming Interface, è un'interfaccia composta di definizioni e protocolli per la creazione e l'integrazione di software applicativi.

CAPITOLO 2. STRUMENTI DI SVILUPPO

- method: Stringa contenente il nome del metodo che la richiesta deve invocare;
- params: Corrisponde ai parametri da passare al metodo sotto forma di array di oggetti;
- id: un valore univoco utilizzato per abbinare la risposta alla richiesta.

Presso questa fonte sono consultabili tutti i metodi del JSON-RPC di Ethereum : [20]. La libreria web3.js comunica con la Blockchain di Ethereum proprio grazie al JSON-RPC, infatti, quest'ultima legge e scrive i dati effettuando richieste JSON-RPC a un nodo Ethereum. Nel progetto di questo elaborato verranno utilizzati principalmente due pacchetti specifici di web3.js:

- web3.eth: pacchetto utilizzato per interagire a tutti gli effetti con la rete Ethereum e gli smart-contract;
- web3.utils: pacchetto contenente diverse funzioni per la conversione di stringhe e numeri in formati specifici.

2.3.3 Metamask

Per sfruttare al meglio la libreria web3.js e semplificarne l'utilizzo c'è bisogno di utilizzare un provider³⁸. Senza di esso un semplice utente avrebbe molta più difficoltà a comunicare con la Blockchain. Il provider scelto per l'elaborato è *Metamask*.

Metamask è un *wallet*(ossia un portafoglio digitale) utilizzato per la custodia dei propri asset (nel nostro caso criptovalute) e per comunicare con applicazioni decentralizzate. Esso può essere utilizzato direttamente dal proprio browser tramite un'estensione, semplificando al massimo la fruizione dei servizi offerti da una Dapp. Una volta installata questa estensione, Metamask istanzierà un oggetto Ethereum alla finestra principale del browser, la quale presenza è verificabile tramite web3.js. A questo punto l'utente potrà eseguire tutte le operazioni che la Dapp gli consente di fare e Metamask gestirà tutte le transazioni associate.

³⁸Anche chiamato Internet service provider (ISP) indica un'organizzazione o un'infrastruttura che offre servizi (inerenti a Internet) agli utenti.

Capitolo 3

Implementazione degli applicativi

3.1 Introduzione e caratteristiche

In seguito alla descrizione degli strumenti di sviluppo utilizzati, introduciamo l'implementazione vera e propria degli applicativi. In questo capitolo verranno mostrate due differenti implementazioni di applicazione decentralizzata, una riguardante lo staking di criptovalute (3.2), nella quale verrà mostrato lo sviluppo front-end dell'applicazione e verrà spiegato nel dettaglio il funzionamento dello staking; l'altra che rappresenterà un fac-simile di exchange decentralizzato (DEX)(3.3), nella quale, oltre a mostrare lo sviluppo front-end dell'applicazione, verrà illustrato il funzionamento di un Automatic Market Maker (AMM), l'interfacciamento con gli smart-contract in Solidity e tutti i funzionamenti intrinseci di un exchange decentralizzato.

3.2 Implementazione dello staking

In questa sezione verrà mostrato lo sviluppo prettamente lato front-end di un applicazione decentralizzata (Dapp) in React. L'applicazione, per quanto possa ancora essere un prototipo, è un esempio di piattaforma per lo staking di criptovalute strutturata sulla rete di test Ropsten. L'interfaccia grafica utilizza una logica intuitiva che permette all'utente una visualizzazione semplificata e una veloce interazione con Metamask (e quindi coi propri cripto asset).

3.2.1 In cosa consiste lo staking?

Lo staking è quel processo che alcune criptovalute, la cui Blockchain utilizza come algoritmo di consenso la Proof-Of-Stake (PoS), permettono di

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

realizzare.

Lo staking di criptovalute consiste in una procedura passiva attraverso cui un possessore di cripto asset (staker) blocca i propri asset per un determinato periodo di tempo; da questa procedura riceverà delle ricompense sotto forma di criptovalute per generare guadagno.

Gli staker ricevono ricompense dallo staking perché aiutano la Blockchain a confermare le transazioni in maniera decentralizzata e aggiungere nuovi blocchi alla catena. Questo processo è molto simile al mining visto in (1.2.5), con la differenza che in una Blockchain utilizzatrice di PoS viene chiamato *forging* (o anche *minting*) e che i miner vengono chiamati *forger* (o anche *validatori*). Esistono diverse tipologie di staking, definiamo un breve elenco dei più usati:

- Il *Pool Staking*, il quale consiste in uno *staking pool*³⁹ nel quale vengono suddivise, in modo del tutto proporzionale, le ricompense tra tutti gli staker;
- Il *Cold Staking* consiste nel processo di staking in un wallet offline, garantendo la sicurezza massima nell'investimento che si vuole fare;
- Lo *Staking Delegato*, invece, consiste nell'affidamento di questa procedura a piattaforme di exchange o altri intermediari. Questi ultimi rendono molto semplici e intuitive le operazioni per mettere in stake le proprie criptovalute, garantendo una grande facilità di utilizzo.

Nonostante la maggior parte delle piattaforme (exchange nel nostro caso) per lo *staking delegato* sfruttano la Proof-Of-Stake (PoS) per generare profitti, la nostra applicazione ha come obiettivo primario quello di incentivare gli utenti a mettere in stake (e di bloccare) i token proprietari del sistema Atmosphere Arc [21] e generare fees dalle varie piattaforme che sono affiliate e supportano il sistema appena citato.

I token proprietari sono chiamati ATMOS e fanno parte dello standard dei token ERC-20⁴⁰.

3.2.2 Interfaccia Grafica dell'applicativo

L'applicazione sviluppata è un prototipo di UI capace di rendere utilizzabile e accessibile lo staking dei token ATMOS per tutti gli utenti possessori. Di seguito viene mostrata l'interfaccia e tutte le sue peculiarità

³⁹Consente a più stakeholder di unire e combinare la propria potenza computazionale in maniera tale da aumentare le proprie possibilità di essere premiati.

⁴⁰Un token ERC-20 (Ethereum request for comment) è uno standard per i token fungibili utilizzato per creare ed emettere smart-contract sulla Blockchain di Ethereum.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

Navbar e Footer

Navbar e Footer sono due componenti essenziali in una pagina web, permettono un elenco organizzato di collegamenti ad altre pagine Web (solitamente pagine interne); permettono una più veloce navigazione all'interno della pagina e una maggiore pulizia nella visualizzazione dei contenuti.

Nel nostro applicativo, la barra di navigazione è una *sideBar* che viene mostrata con un'animazione all'onClick dell'icona del menu. Di seguito viene mostrato il codice CSS e la logica JSX per rendere possibile l'animazione.

```
1 /*JSX*/
2 const showSidebar = () => {
3     setSidebar(!sidebar)
4 }
5 return (
6 /*[...])
7 <nav className={sidebar ? "nav-menu active" : "nav-menu"}>
8     <ul className="nav-menu-items" onClick={showSidebar}>
9         <li className="nav-menu-toggle">
10             <Link to="#" className="bars">
11                 <AiIcons.AiOutlineClose/>
12             </Link>
13         </li>
14     {SidebarElements.map((item, index) => {
15         return (
16             <li key={index} className={item.cssName}>
17                 <Link to={item.path}>
18                     {item.icon}
19                     <span style={{marginLeft: 15}}>
20                         {item.title}
21                     </span>
22                 </Link>
23             </li>
24         );
25     ))}
26     </ul>
27 </nav>
28 /*[...])
29 /*CSS*/
30 .nav-menu{
31     z-index: 1;
32     background-color: #060b26;
33     width: 250px;
34     height: 100vh;
35     display: flex;
36     justify-content: center;
37     position: fixed;
38     top: 0;
39     left: -100%;
40     transition: 850ms;
41 }
```

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

```
42 | .nav-menu.active{  
43 |   left: 0;  
44 |   transition: 350ms;  
45 | }  
                                         —navbarCode
```

L’animazione avviene tramite l’assegnazione dinamica di stile CSS, passando dalla classe `nav-menu` a `nav-menu active`(come mostrato a riga 15) finchè l’utente non interagisce con l’icona del menu, ossia, fin quando la variabile “sidebar” viene modificata attraverso la funzione “showSideBar” a riga 2. Il contenuto della Sidebar viene generato attraverso il mapping⁴¹ degli oggetti restituiti dalla classe `SidebarElements.js`, la quale definisce i bottoni che la navbar permetterà di selezionare; la strategia applicativa appena descritta permette un rendering molto efficiente. Viene mostrata in figura 3.1 la visualizzazione della `sidebar` appena descritta.

Siccome React impiega un DOM virtuale, che traccia le modifiche e aggiorna solo gli elementi variati, garantisce un aggiornamento efficiente ed un’ottima user experience.

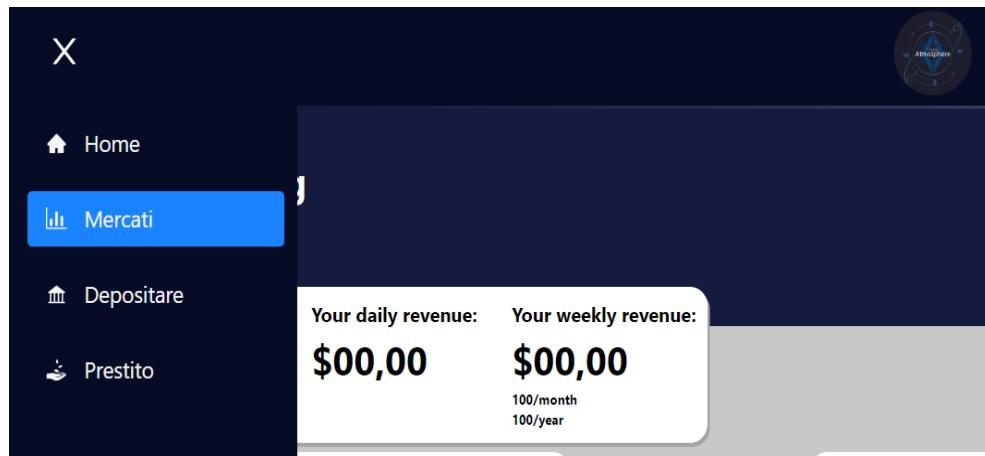


Figura 3.1: Sidebar dell’applicativo.

Per quanto riguarda il Footer dell’applicazione, vengono mostrate piccole sezioni per le informazioni della compagnia, FAQ, supporto e icone per la renderizzazione ai rispettivi canali social. Viene mostrato in figura 3.2.

I due elementi appena mostrati vengono definiti nelle componenti `NavBar.js` e `Footer.js` con il relativo stile CSS importato al loro interno.

⁴¹La mappatura dei dati consiste nell’applicare una funzione a tutti i membri di un elenco o di una struttura simile. L’uso della mappatura può aiutarti a regolare l’intervallo dei valori o a preparare i valori per particolari tipi di analisi.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

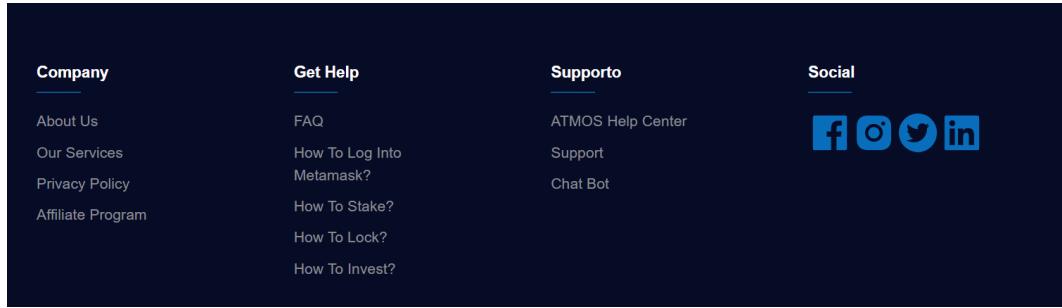


Figura 3.2: Footer dell'applicativo.

Pagina principale

La pagina principale dell'applicazione consiste in tanti Card che mostrano diverse informazioni all'utente. Queste informazioni devono essere prelevate, elaborate e mostrate in maniera adeguata al fruitore dei servizi dell'ipotetico applicativo finale che verrà sviluppato su questo prototipo. Prima di presentare le interfacce dell'applicazione introdotta in (3.2), mostriamo la struttura delle componenti e della loro gerarchia in figura 3.3. Alla radice della no-

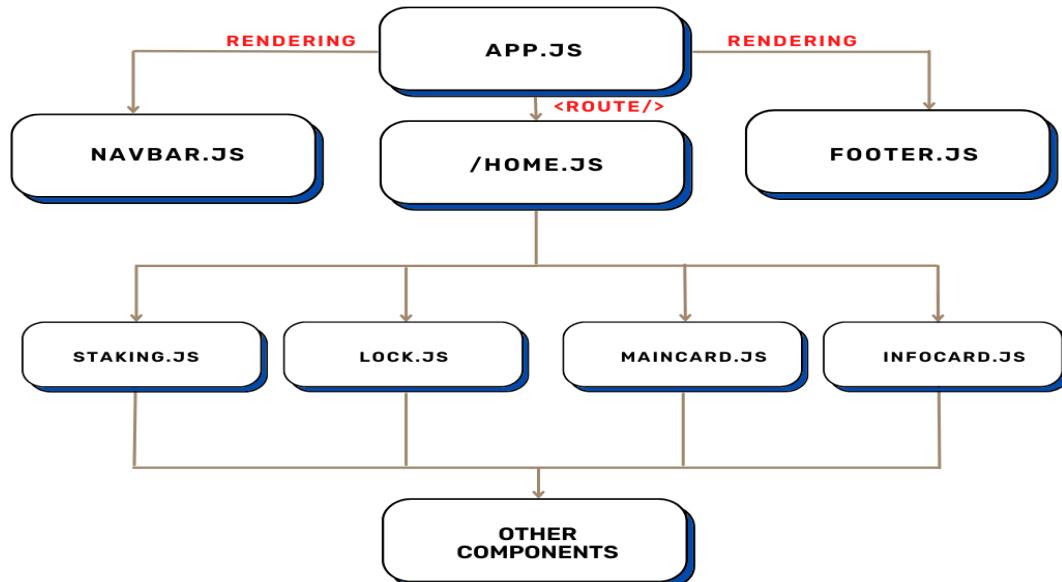


Figura 3.3: Struttura delle componenti dell'applicativo riferito allo staking.

stra applicazione, come di consuetudine per ogni applicazione in React, c'è **App.js**, la quale è collegata (attraverso meccanismi interni approfonditi in 3.3.3) alle componenti successive dell'albero. Al di sotto della radice (oltre

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

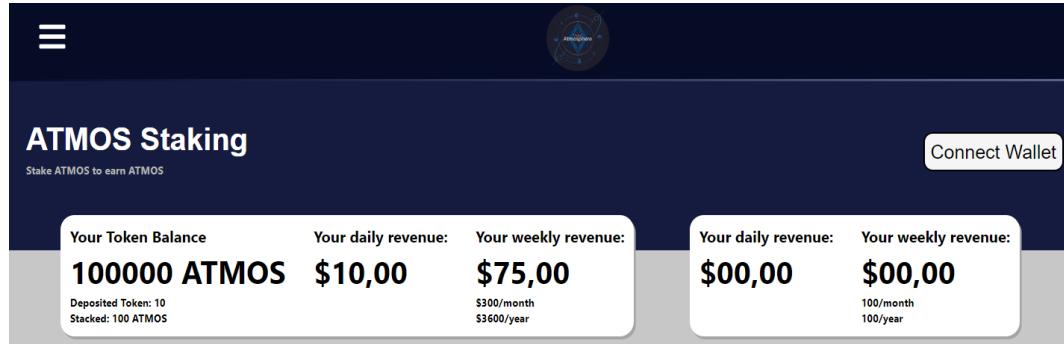


Figura 3.4: Card che mostrano il balance e le revenue dell’utente collegato.

alle componenti sempre presenti `Navbar.js` e `Footer.js`) c’è la rotta riferita ad `Home.js` la quale possiede a sua volta altre 4 componenti principali in cui sono presenti i Card principali per la visualizzazione dei dati: `InfoCard.js`, `Staking.js`, `Lock.js`, `MainCard.js`.

I primi Card, mostrati in figura 3.4, fanno parte della componente `MainCard.js`, comunicano all’utente la loro disponibilità dei token ATMOS (i quali vengono conteggiati attraverso Metamask), le revenue giornaliere, mensili e annuali, i relativi token depositati e messi in stake. Le informazioni in questione possono essere mostrate soltanto agli utenti connessi all’applicazione. La logica della **Connessione a Metamask** verrà mostrata nella parte dell’elaborato relativa all’implementazione dell’exchange.

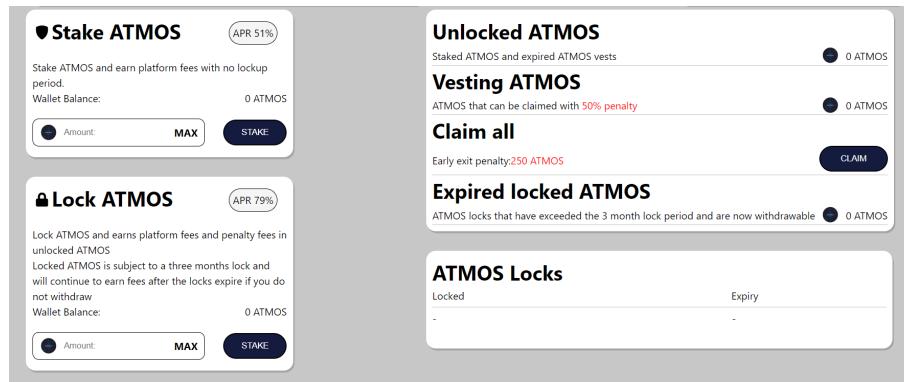


Figura 3.5: Card che mostrano i form per la messa in stake e in lock dei token.

Invece, per quanto riguarda il corpo vero e proprio dell’applicativo, i Card mostrati in figura 3.5 sono composti da form per mettere in stake e

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

in lock⁴² quantitativi variabili di token ATMOS. Questi Card appartengono alle componenti **Staking.js**, **Lock.js** e **InfoCard.js**, i quali, attraverso il codice JSX e il CSS specifico importato nelle componenti, renderizza gli elementi descritti.

Una volta inserito l'ammontare di token da mettere in stake o in lock, l'utente dovrà cliccare sul bottone riferito all'operazione corrispondente; si dovrà poi confermare l'operazione attraverso l'approvazione dei token che si vogliono utilizzare. Come è stato accennato in 2.3.3, Metamask gestisce tutte le comunicazioni con la Blockchain integrando la visualizzazione per la conferma di operazioni (come quella dell'approvazione nel nostro caso specifico) in pop-up che vengono visualizzati all'utente in maniera molto veloce e sintetica. Si mostra in figura 3.6 un esempio di chiamata all'approve dei nostri token ATMOS.

L'operazione di approvazione consiste nell'impostare un valore (nel nostro caso l'importo inserito nei form) come indennità di spesa sui token del chiamante. La funzione di approve appartiene all'ABI standard dei token ERC-20 e richiede come parametri il valore appena accennato e l'address del token ERC-20 per cui si vuole fare l'approvazione. Con questa funzione, si può assegnare ad un altro indirizzo i permessi di spendere i token. La limitazione sui token che si approvano permette di evitare exploit o l'azzeramento dei fondi da parte di malintenzionati. Come anticipato, React utilizza

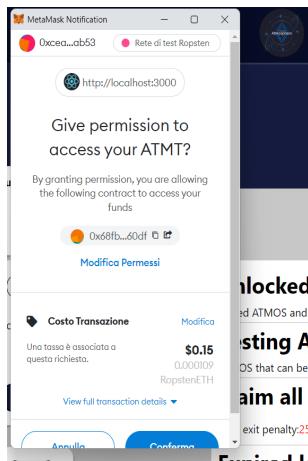


Figura 3.6: Esempio di approvazione dei token ATMOS tramite Metamask.

il Virtual DOM per apportare modifiche o per effettuare controlli specifici; sfruttiamo questo vantaggio per controllare se l'utente possiede abbastanza

⁴²Il lock di token si riferisce a un periodo di tempo specifico in cui i token non possono essere inoltrati o scambiati. Tipicamente, viene utilizzato come strategia preventiva per mantenere un valore stabile a lungo termine di un particolare asset.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

token oppure se è stata effettuata l'approvazione prima di un'operazione. La visualizzazione del messaggio viene mostrata in figura 3.7.

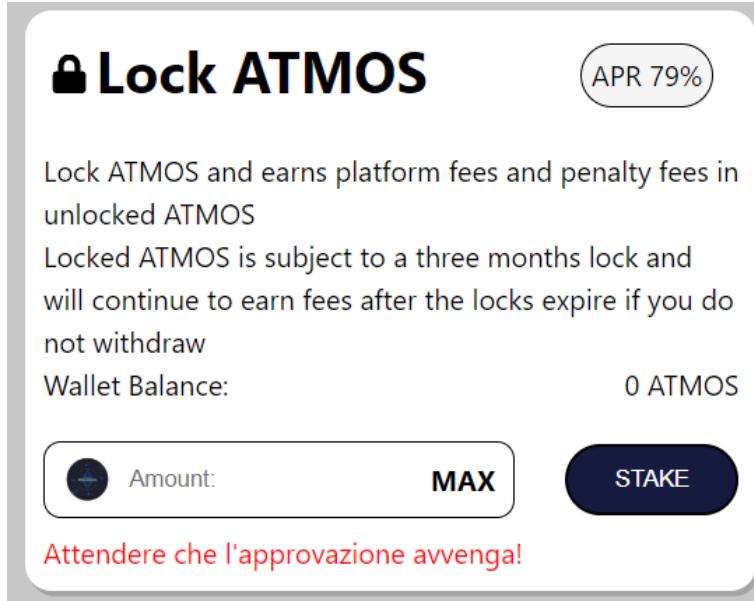


Figura 3.7: Messaggio di mancata approvazione dei token visualizzato nel Card.

Il controllo in questione deve essere fatto finché un ipotetica operazione *asincrona*⁴³ (che sia un'approve oppure la richiesta della conferma di un'operazione attraverso Metamask) non venga terminata o confermata dall'utente. Così facendo, si simula l'handling dei messaggi di errore, così come si farebbe con JQuery⁴⁴.

3.3 Implementazione dell'exchange

La maggior parte del lavoro di questo elaborato è stato dedicato alla realizzazione di un exchange decentralizzato (DEX), anch'esso strutturato sulla rete di test Ropsten su Blockchain Ethereum.

Prima di indentrarci nell'implementazione, c'è bisogno di introdurre il funzionamento dei DEX e, quindi, di un Automatic Market Maker (AMM). Verrà mostrato il motivo per il quale i DEX hanno rivoluzionato il mondo del-

⁴³Un'operazione asincrona è un'unità di lavoro che funziona separatamente dal thread principale, notificando, successivamente, quando avrà finito il lavoro.

⁴⁴Libreria JavaScript progettata per semplificare l'attraversamento e la manipolazione dell'albero DOM HTML, nonché la gestione degli eventi, l'animazione CSS e Ajax.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

la finanza decentralizzata (DeFi), definendo tutte le logiche, le caratteristiche e i servizi da essi offerti.

3.3.1 Exchange Decentralizzato (DEX)

La finanza decentralizzata (DeFi) ha offerto al mondo un modo notevolmente migliore di fare trading e di guadagnare entrate passive, in questo caso tramite le criptovalute. Prima che esistesse la DeFi, il trading veniva condotto dagli *exchange centralizzati* (CEX), i quali vengono creati e organizzati da organizzazioni centralizzate che fungono da intermediari tra gli acquirenti e i venditori. Essi mantengono il controllo sulle chiavi private (evitando agli utenti l'onere di gestirle), i parametri di trading e le informazioni relative all'utente; memorizzano i dati di tutti gli ordini di acquisto e di vendita per un determinato asset che l'utente effettua, archiviandoli all'interno di un database centralizzato. Il controllo e la gestione degli elementi appena citati, rende questi exchange molto facili da utilizzare, anche per utenti nuovi nel settore.

Uno dei più grandi svantaggi dei CEX è quello legato alla sicurezza: lasciare che l'exchange gestisca la chiave per il portafoglio significa che qualsiasi risorsa al suo interno non è veramente tua; di fatto se l'exchange venisse violato, o addirittura avvenisse un fallimento dell'azienda, tutto quello che si possiede nel proprio portafogli andrebbe perso. In poche parole, i CEX sono un compromesso per gli utenti che non vogliono avere l'onere, come anticipato, di gestire le proprie chiavi.

Un'exchange decentralizzato (DEX), invece, si porta con sé le caratteristiche derivanti dalla DeFi, preservando l'etica crittografica e rendendolo uno strumento sicuro e privo di intermediazione. Senza un'entità centrale non esisterà una piattaforma in cui depositare dei fondi, l'utente dovrà soltanto collegare il proprio portafoglio al DEX utilizzando le proprie chiavi private. Questo ovviamente implica l'onere di gestire con cura e con attenzione le proprie chiavi; d'altra parte, rende le risorse in tutto e per tutto controllabili solo dall'utente e non da entità centrali.

I DEX, inoltre, non interagiscono con il denaro fiat⁴⁵, quindi non necessitano la conformità a KYC⁴⁶. Grazie a questo, la tua privacy, le tue informazioni personali e i tuoi dati sensibili non possono essere visti o hackerati da nessuna persona (i dati non fanno parte di nessuna istituzione o ente centrale). Esistono due differenti tipi di DEX:

⁴⁵ Con denaro fiat si intende una valuta emessa dal governo che non è supportata da un bene fisico, come oro o argento, ma piuttosto dal governo che l'ha emessa.

⁴⁶ acronimo di Know Your Customer (letteralmente: “conosci il tuo cliente”), è l'insieme di procedure che devono essere attuate da alcuni istituti e professionisti per obbligo di legge. Queste procedure servono per acquisire dati certi e informazioni sull'identità dei loro utenti e clienti.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

- Basato sugli *Order-book*: i quali utilizzano un algoritmo (invece di una piattaforma centrale) per trovare e instradare gli scambi tra i singoli utenti. Gli smart-contract registrano gli scambi sulla Blockchain per riflettere gli asset che si spostano tra acquirenti e venditori. Non esiste un mercato vero e proprio, nel mezzo si interpone solo un algoritmo, rendendo comunque il servizio decentralizzato;
- Gli *Automatic Market Maker* (AMM): la nostra applicazione si posiziona in questa categoria di DEX, nel paragrafo successivo (3.3.2) si approfondisce il tema in questione.

3.3.2 Automatic Market Maker (AMM)

Gli Automatic Market Maker (AMM) sono dei sistemi basati su Blockchain che permettono il trading *automatizzato* attraverso scambi di criptovalute decentralizzati, il tutto regolato dagli smart-contract. In un contesto di questo tipo, il codice degli smart-contract è legge, il quale elabora tutte le transazioni automaticamente, senza fare affidamento su richieste di acquisto/vendita di terze parti per i token scambiati. Prenderemo in esempio il protocollo di Uniswap [22] per la spiegazione tecnica del funzionamento di questi sistemi.

Pool di liquidità

Tra gli smart-contract più importanti di un AMM ci sono quelli riferiti al *Pool di liquidità*. Un *Pool di liquidità*, come mostrato nella rappresentazione 3.8 in [23]



Figura 3.8: Rappresentazione grafica di un Pool di Liquidità.

non è altro che una raccolta di token crittografici, che viene utilizzato per rendere possibile il trading tramite un algoritmo che definisce i prezzi dei token in base al rapporto variabile dei token forniti; infatti, come definito in [23]: “Ogni pool di liquidità è una sede di negoziazione per una coppia di token ERC-20.[...]”.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

Ogni *Pool di liquidità* è composto da due token (nel caso di Uniswap da due token ERC-20), ecco perché i pool di liquidità sono anche chiamati coppie (o Pair). Quando viene creato un nuovo contratto Pool i saldi di ciascun token sono uguali a zero, saranno i Liquidity Provider (LP, letteralmente fornitori di liquidità) o market maker, che forniranno la liquidità ai pool sotto forma di asset digitali. I primi fornitori del pool saranno anche coloro che determineranno il prezzo iniziale del Pool. I Liquidity Provider (LP) fanno parte della prima parte, raffigurata in figura 3.9.

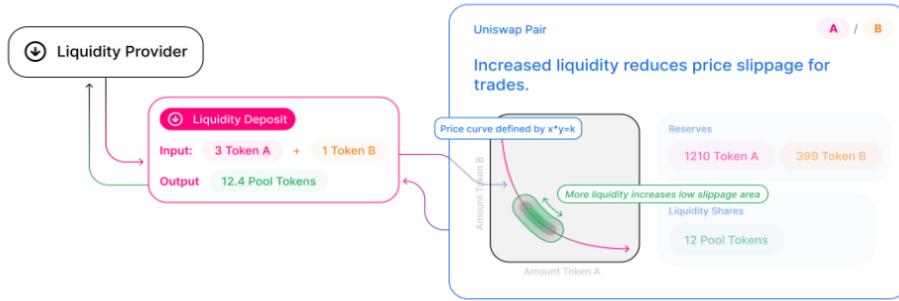


Figura 3.9: Rappresentazione deposito liquidità da parte dei Liquidity Provider (LP).

Negli AMM, è importante notare come i Pool con grandi quantità di trade attivi (e quindi grande liquidità) prosperino sulla rete, mentre i Pool con bassa liquidità causano enormi impatti sui prezzi e slittamenti vari, rendendoli, quindi, inefficaci. Inoltre, per depositare fondi in un pool, è necessario fornire una stessa quantità di liquidità per token.

Questi vincoli nascono affinché tutti gli utenti possano scambiare qualsiasi token in qualsiasi momento; il Pool deve sempre disporre di quantità sufficienti di entrambi i token della coppia del Pool, pertanto, ogni DEX che opera sul modello AMM è interessato ad avere una liquidità più grande possibile. Per ottenere una grande liquidità nei Pool, ogni AMM incentiva gli utenti a depositare token nei propri Pool, tramite il concetto di *yield farming* (o liquidity mining).

L'idea dello *yield farming* è quella di ricompensare gli utenti fornitori di liquidità con delle commissioni speciali, chiamate *commissioni LP* (o LP token), ossia commissioni di negoziazione distribuite in maniera proporzionale alla quantità di liquidità apportata, rappresentante una parte di proprietà del Pool. Se un Liquidity Provider (LP) vuole rimuovere liquidità e riacquisire i propri asset, deve bruciare i suoi LP token per recuperare gli asset (ossia i token originali).

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

Token Swaps

Dall'altra parte dell'AMM, come mostrato in figura 3.10, c'è la sezione relativa ai Trader. I trader rappresentano tutti quegli utenti che desiderano

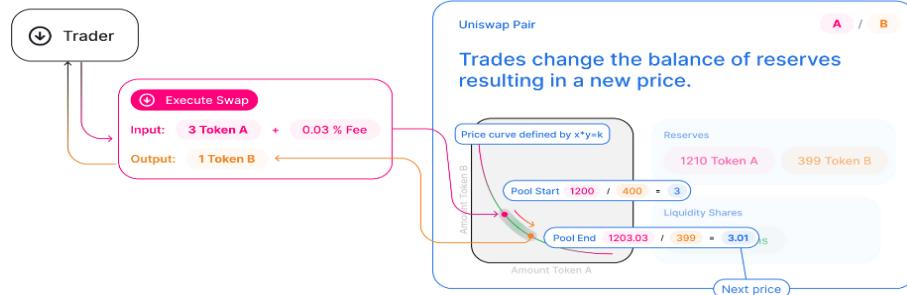


Figura 3.10: Sezione relativa al trading di token.

scambiare token per altri token e pagare il prezzo in base alla riserva di liquidità (mostrata in figura 3.8) nel contratto. Come anticipato in ([Pool di liquidità](#)), ogni coppia di token è supportata da un Pool di liquidità, il quale, ricordiamo, non è altro che uno smart-contract che definisce tutte le operazioni per quanto riguarda le operazioni di deposito e di prelievo, oltre che utilizzato per detenere i saldi dei due token. Per assicurarsi che il rapporto tra le attività nei pool di liquidità rimanga il più equilibrato possibile eliminando eventuali discrepanze nei prezzi delle attività nei pool, gli AMM utilizzano equazioni matematiche preimpostate. Nel caso di Uniswap, come definito in [24], viene utilizzata la “*formula del prodotto costante*” ossia:

$$x \cdot y = k, \text{ dove :}$$

- x rappresenta il valore del primo asset (token A);
- y rappresenta il valore del secondo asset (token B);
- k , invece, è una costante.

Sostanzialmente Uniswap, attraverso questa formula, mantiene sempre un valore costante, uguale nel tempo, per la moltiplicazione del prezzo del token A col token B. Prendiamo in esempio un ipotetico Pool di liquidità tra un token A e un token B: se il token A viene acquistato dai trader, il token B verrà aggiunto al Pool, rimuovendo il token A da esso. Questa operazione fa diminuire la quantità del token A all'interno del Pool, il che, a sua volta, fa aumentare il suo prezzo (proprio per soddisfare l'effetto di bilanciamento della formula sopra citata). Mentre il prezzo del token A aumenta di prezzo, il prezzo del token B diminuisce a sua volta perché aggiunto al Pool.

Routing

Spesso i trader hanno bisogno di scambiare tipologie di asset per i quali non esiste un Pool di liquidità; la creazione di un Pool potrebbe essere molto costoso per alcuni trader, di conseguenza, per far fronte a questo problema, è nato il concetto del *Routing*. Il *Routing* è quella procedura che permette ai trader (o utenti della Dapp fruitori dei servizi) di scambiare asset anche quando non esiste un Pool di liquidità tra una coppia di asset su cui vogliono operare. Questa procedura consiste nello scandagliare tutti i Pool di liquidità che esistono tra i due token della coppia che si vuole scambiare e scegliere la *Custom Path* (la rottura) da seguire per ottenere il miglior prezzo possibile per l'acquirente. Lo smart-contract che si occupa di effettuare l'operazione di Routing è il *Router Contract* (che approfondiremo nei paragrafi successivi). Il contratto in questione, esegue queste operazioni totalmente in background, in un'unica transazione di scambio, in maniera tale che l'utente non si accorga di nulla e renda le operazioni molto più semplificate. Un esempio di rottura tra Pool di liquidità viene mostrato nella figura 3.11.

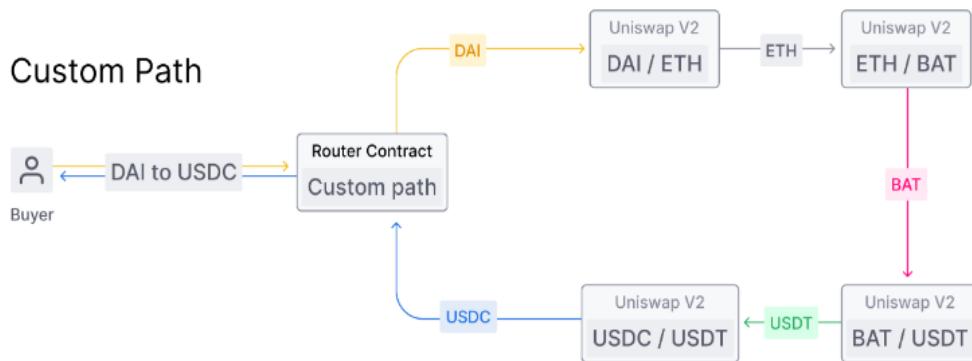


Figura 3.11: Esempio di Routing tra più Pool di liquidità.

3.3.3 Interfaccia Grafica dell'applicativo

La maggior parte del lavoro correlato a questo elaborato consiste, come già anticipato, nell'implementazione di un exchange decentralizzato (DEX) distribuito sulla rete di test Ropsten di Ethereum. L'applicativo in questione consta di due pagine principali:

- La pagina relativa allo *Swap* dei token ERC-20 presenti sulla rete di Ethereum;
- La pagina relativa ai *Pool di liquidità* per ogni coppia di token su cui si vuole operare.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

Queste due pagine rappresentano le parti nelle quali si concentra tutta la logica dell'exchange e l'interazione vera e propria con gli smart-contract sviluppati in Solidity (approfondiremo la logica in 3.3.4). Mostriamo ora l'interfaccia grafica dell'applicativo, la quale è stata pensata per essere user-friendly, facile da utilizzare e intuitiva il più possibile.

Struttura delle componenti

Prima di mostrare le varie interfacce implementate, definiamo la struttura delle componenti che abbiamo scelto per la nostra applicazione React. Si è sintetizzata la gerarchia delle componenti nella figura 3.12.

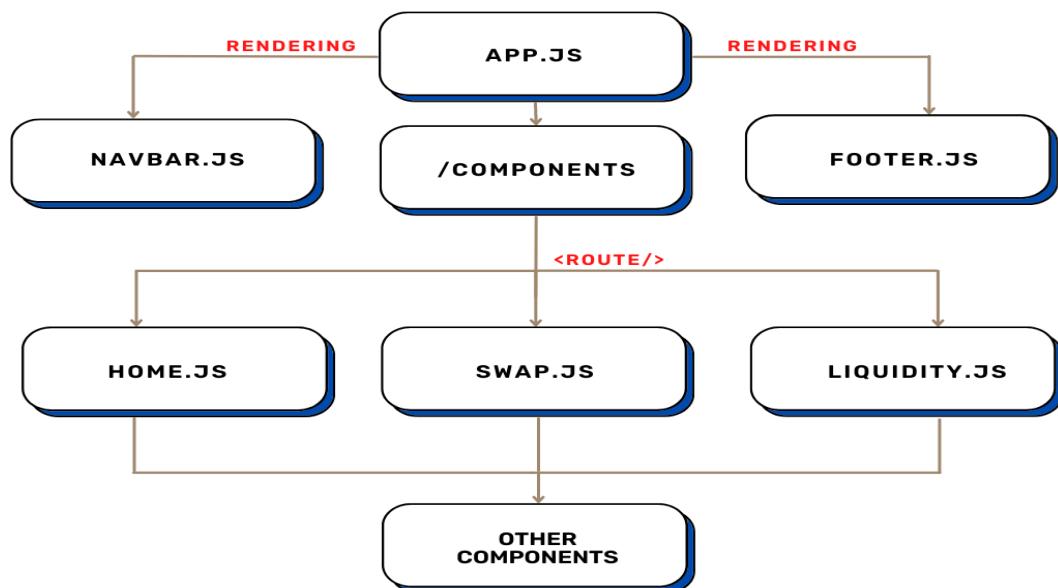


Figura 3.12: Struttura delle componenti dell'applicativo riferito al DEX.

Come si può notare, alla radice della nostra gerarchia c'è la componente **App.js**, la quale esegue il rendering dell'interfaccia utente globale dell'app e configura tutte le *variabili di stato* necessarie al corretto funzionamento delle procedure eseguite nelle altre componenti renderizzate da quest'ultima. Per il corretto rendering delle componenti sottostanti ad **App.js** si utilizza la libreria **react-router**, in quanto un'app React di base non consente l'instradamento delle pagine. Una volta scaricata la libreria tramite il comando **npm install react-router –save**, è possibile definire diverse “rotte” per la nostra applicazione (come mostrato in figura 3.12). Di seguito si mostra il codice JSX del rendering delle componenti principali della gerarchia (notare come queste siano solo le principali, ogni componente possiede altre sotto-

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

componenti per la gestione dinamica delle proprie funzionalità specifiche):

```
1  return (
2    <>
3      <Router>
4
5        {/*Navbar component*/}
6        <Navbar isConnected={isConnected}
7          currentAccount={currentAccount}
8          web3={web3} showCard={showCard}
9          active={active}/>
10
11       {/*ROTTE DELLA PIATTAFORMA*/}
12       <Switch>
13
14         <Route path='/swap'
15           render={()=> <Swap web3={web3}
16             router={routerContract} factory={factoryContract}
17             isConnected={isConnected} showCard={showCard}
18             currentAccount={currentAccount}/>}
19           />
20
21         <Route path='/liquidity'
22           render={()=> <Liquidity web3={web3}
23             router={routerContract} factory={factoryContract}
24             isConnected={isConnected} showCard={showCard}
25             currentAccount={currentAccount}/>}
26           />
27
28       </Switch>
29
30       {/*Footer component*/}
31       <Footer/>
32
33     </Router>
34   </>
35 );
```

—App.js

Come è visibile nel codice JSX di [App.js](#), le componenti [Navbar.js](#) e [Footer.js](#) sono sempre presenti, a prescindere dalla rotta, per ogni rendering della pagina. Questo perché la Navbar e il Footer, in generale in un'applicazione web, sono due elementi essenziali che, solitamente, si ripetono in ogni pagina. Nella riga 19 e 26 ,rispettivamente, vengono definite le rotte per le componenti principali dell'applicazione sviluppata:

- La componente [Swap.js](#) la quale interfaccia sarà approfondita in ([Swap.js](#));
- La componente [Liquidity.js](#) la quale interfaccia sarà approfondita in ([Liquidity.js](#)).

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

Prima di approfondire le due componenti sopra citate, nel paragrafo successivo verrà spiegato come avviene la connessione al nostro wallet digitale (e quindi la possibilità di comunicare in maniera molto semplificata con la Blockchain), le funzioni chiamate attraverso la libreria *web3.js* e le relative interfacce grafiche per l'autenticazione.

Connessione a Metamask

La connessione al wallet di Metamask (e il relativo accesso ai propri token), viene gestita tramite la libreria *web3.js*; a livello grafico, viene mostrato un bottone all'interno della Navbar 3.13, il quale, all'*onClick*, tramite delle funzioni CSS, rende il background sfocato per dirigere l'attenzione dell'utente su un Card (renderizzato dalla componente **ConnectionCard.js**) che visualizza graficamente diversi provider ai quali ci si può collegare. Nel nostro caso prenderemo in esempio l'utilizzo di Metamask.

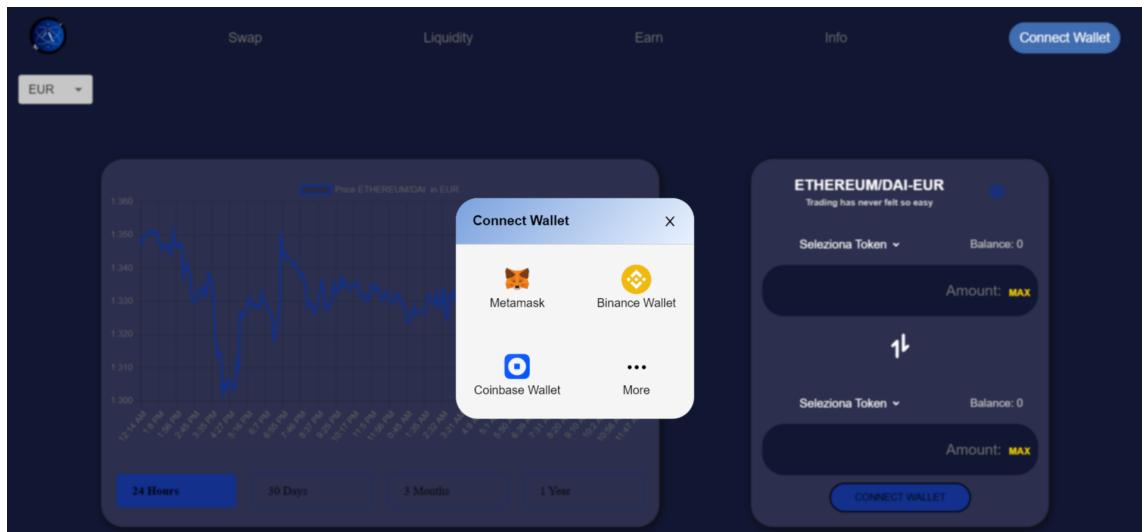


Figura 3.13: Card per la scelta del provider visualizzato all'*onClick* del bottone “Connect Wallet”.

Una volta scelto il wallet, viene eseguita la funzione **onLogin**: il codice viene mostrato di seguito.

```
1 const onLogin = async (provider) => {
2     const web3= new Web3(provider)
3     const accounts = await web3.eth.getAccounts();
4     const chainId = await web3.eth.getChainId();
5     if (accounts.length === 0) {
6         console.log("Please connect to MetaMask!");
7     } else if (accounts[0] !== currentAccount) {
8         setProvider(provider);
```

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

```
9      setWeb3(web3);
10     setCurrentNetwork(chainId);
11     setCurrentAccount(accounts[0]);
12     setIsConnected(true);
13     window.localStorage.setItem('IS_CONNECTED', accounts[0]);
14
15     /*Create local contracts copy */
16     const factory=SetAtmosFactoryContract(web3)
17     setFactoryContract(factory)
18
19     const router=SetAtmosRouterContract(web3)
20     setRouterContract(router)
21   }
22 };
```

–onLogin

Nella funzione `onLogin()`, viene utilizzata una funzione del pacchetto `web3.eth`: la funzione `getAccounts()`, che non fa altro che restituire l'elenco degli account controllati dal nodo, come definito in [19]. Nella riga 3 viene assegnato l'oggetto restituito dalla funzione; se questo oggetto esiste significa che l'utente ha effettuato con successo la connessione a Metamask, altrimenti si controlla (nella riga 7) se l'account con il quale si è fatto l'accesso è diverso dall'account corrente: in poche parole si verifica se un utente ha già effettuato l'accesso alla Dapp. Se sì, la funzione termina la sua esecuzione perché i dati per quell'utente sono già stati settati, se no, invece, viene confermato che è il primo accesso dell'utente in questione e si settano tutte le variabili utili per il corretto funzionamento delle funzionalità successive. Da notare come le variabili vengono settate attraverso lo *useState Hook* di React, definito in [18]. La schermata di caricamento col relativo inserimento della propria password, viene visualizzata finché non si completa la riga 3 della funzione `onLogin()` (il comando `await` viene utilizzato per attendere una *Promise*⁴⁷ e ottenere il suo valore), la quale attiva l'estensione Metamask del browser web, mostrando un pop-up con un form per l'inserimento dei dati relativi all'accesso. Viene mostrata la schermata nella figura 3.14.

Una volta effettuato l'accesso a Metamask, si visualizza l'indirizzo pubblico del proprio wallet direttamente nella Navbar dell'applicazione, in maniera tale da rendere la gestione dei propri address più semplice e intuitiva.

Componente Swap.js

La componente **Swap.js** è la parte del nostro exchange che si occupa dello Swap dei token ERC-20 della rete Ethereum. L'interfaccia grafica della componente è composta da un grafico animato, strutturato grazie all'ausilio di una libreria React per la composizione di grafici: la libreria `react-chartjs`

⁴⁷L'oggetto Promise rappresenta l'eventuale completamento (o fallimento) di un'operazione asincrona e il suo valore risultante.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

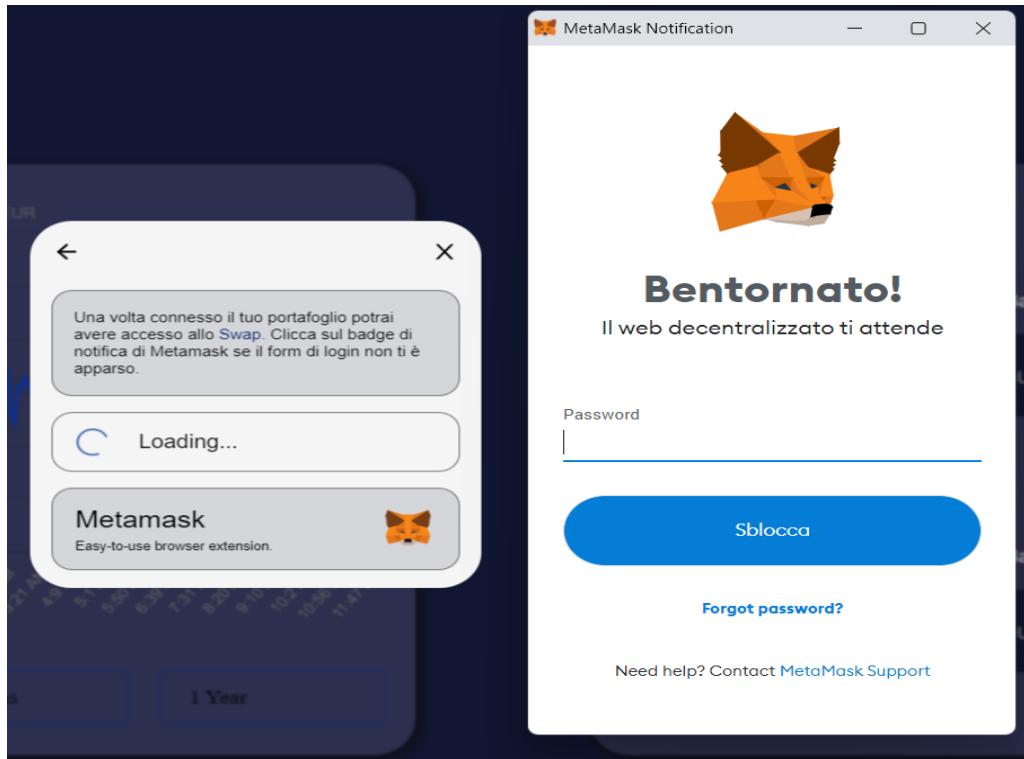


Figura 3.14: Schermata di caricamento in attesa che l’utente inserisca i dati di accesso.

2, installata tramite il comando `npm install react-chartjs-2`. Il grafico in questione si tratta di un *Line Chart*, il quale riceve dei dati in input da parte di una API di *CoinGecko* [25] che ottiene i dati storici (nome, prezzo, mercato, statistiche) in una determinata data per il singolo token. Il codice relativo alla chiamata dell’API viene mostrato in **APIcode**. La chiamata all’API in questione è stata riscritta come una *arrow function*⁴⁸ rinominata “*HistoricalChart*” nella componente **API.js**. Questa funzione richiede tre parametri in input, rispettivamente:

- L’id del token (ossia il nome del token) scelto per cui si vuole visualizzare il prezzo nel corso del tempo;
- I days (giorni), ossia il range di tempo per ricevere l’istantanea dei dati;

⁴⁸Scrittura alternativa (più pulita) di una funzione tradizionale. Questa funzionalità è stata introdotta nella versione ES6 di JavaScript.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

- La currency, ossia la valuta con cui si vuole visualizzare i prezzi (nella applicazione vengono gestiti i prezzi in Euro e in Dollari).

Da notare come nella funzione *HistoricalChart* siano stati utilizzati i *Template Literals*, i quali permettono di creare stringhe interpolate, ossia stringhe con l'aggiunta dinamica di elementi all'interno di esse.

```
1  /*Componente API.js, la quale definisce tutte le chiamate API
2   che si vogliono utilizzare per inviarle al server web*/
3
4
5 export const HistoricalChart = (id, days = 365, currency) =>
6   `https://api.coingecko.com/api/v3/coins/${id}/market_chart?
7   vs_currency=${currency}&days=${days}`;
8
9
10 import { HistoricalChart } from "../config/API";
11 const fetchHistoricData = async () => {
12   const dataTokenA = (await axios.get
13     (HistoricalChart(tokenA ? tokenA.id : "ethereum", days,
14     currency))).data;
15   setHistoricDataTokenA(dataTokenA.prices);
16 }
17
18 useEffect((() => {
19   fetchHistoricData();
20 }, [days, tokenA, tokenB, currency]))
```

—APIcode

Per quanto riguarda, invece, la chiamata GET effettiva da mandare al server, si sfrutta un'altra libreria Javascript che ci permette di gestire le richieste effettuate tramite il protocollo HTTP e di connetterci alle API: la libreria **axios**, installabile tramite il comando **npm i axios** (la chiamata `axios` viene definita nella riga 12 di **APIcode**). Sempre in **APIcode**, alla riga 17, viene utilizzato un'altro hook molto importante di React chiamato *useEffect*, il quale, come si dice in [18] :“consente di eseguire effetti collaterali nei componenti”; nel nostro caso, viene utilizzato per chiamare la funzione `fetchHistoricData()` ogni volta che gli elementi presenti nelle *dependency*⁴⁹ dell'hook vengono modificati. Questo meccanismo ci permette di rendere l'interfaccia utente molto dinamica, facendo sì che il grafico (in questo caso specifico) venga aggiornato ad ogni azione dell'utente, permettendogli una veloce e intuitiva comunicazione con l'applicazione. Viene mostrata in figura 3.15 l'implementazione del grafico appena citato.

Come è possibile notare, il *Line Chart* è una sorta di griglia sulla quale è tracciata una linea colorata che definisce l'andamento del prezzo nel corso

⁴⁹Un array di dipendenze richiesto facoltativamente dallo *useEffect*. L'hook esegue il suo contenuto solo se le dipendenze sono cambiate tra i rendering.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI



Figura 3.15: Line Chart di esempio della coppia Ethereum/Dai-token.

del tempo. Il grafico possiede due serie di label: nella parte inferiore e nella parte laterale; nella parte laterale vengono definite le righe corrispondenti al prezzo della coppia tokenA/tokenB, mentre, nella parte inferiore, vengono tracciate le colonne, ognuna delle quali, corrisponde ad un determinato giorno. A seconda del bottone cliccato tra i quattro selezionabili nella parte sottostante, il grafico cambia forma in maniera del tutto animata, definendo una nuova traccia. Inoltre, se si passa col mouse su un punto del grafico è possibile visualizzare il prezzo in quella determinata data (o orario).

La coppia di token mostrata nel grafico viene scelta attraverso il Card implementato per l'operazione di swap vera e propria. Il Card in questione (mostrato in figura 3.16) ci permette di selezionare la coppia di token attraverso un *dropdown menu* di monete(o token) che si apre facendo click sull'etichetta “Seleziona Token”. Verrà inoltre mostrato il *balance* (saldo) dello specifico token scelto andando a controllare le disponibilità nel wallet dell'account collegato alla nostra Dapp; questa procedura di verifica del balance viene effettuata attraverso il comando:

```
await tokenContract.methods.balanceOf(props.currentAccount).call()
```

dove *tokenContract* è l'istanza di un oggetto `web3.eth.Contract` rappresentate il token che è stato selezionato(approfondiremo l'interazione coi contratti dei token ERC-20 in 3.3.4), *balanceOf* è un metodo definito nell'ABI del contratto di un token ERC-20 che richiede come unico parametro

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

l'address del possessore di questi token (nel nostro caso abbiamo inserito il *currentAccount*, ossia l'address dell'account connesso al momento).

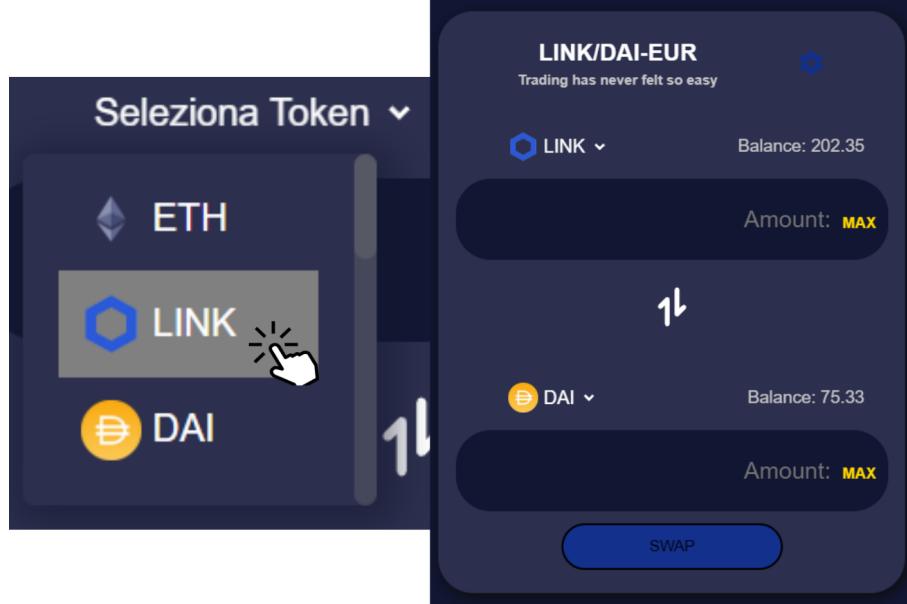


Figura 3.16: Selezione token per il Card relativo allo Swap.

Dal momento che l'EVM non supporta i numeri decimali, bisogna gestire il formato numerico con il quale, il dato token, è stato specificato alla sua creazione. Il formato è verificabile tramite la funzione `decimals()` per ogni token ERC-20. Esistono token con differenti formati, di conseguenza, dal front-end bisogna gestire questa differenziazione per rendere leggibile i dati all'utente; nella nostra applicazione gestiremo i token con unità decimali più diffuse (dai 6 ai più famosi 18 decimali), di seguito viene mostrato il codice per la gestione delle conversioni ([TokenConversion](#)).

```
1 /*Import dell'ABI standard di un token ERC-20*/
2 import ERC20abi from "contracts/ERC20abi.json"
3 const setTokenAContract = async (token) => {
4     if(token.address != null) {
5         const tokenContract =
6             new web3.eth.Contract(ERC20abi, token.address)
7         const val =
8             await tokenContract.methods.balanceOf(props.
9             currentAccount).call()
10        if (token.decimals === 18) { /*18 decimals token*/
11            const bal=web3.utils.fromWei(val, "ether")
12            setBalance(bal)
13        }
14        if (token.decimals === 8) { /*8 decimals token*/
15            const bal=web3.utils.fromWei(val, "gwei")
```

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

```
15         setBalance( bal )
16     }
17     if (token.decimals === 6) { /*6 decimals token*/
18       const bal=web3.utils.fromWei(val , "mwei")
19       setBalance( bal )
20     }
21   } else{ /*Se e' stato selezionato Ethereum nel menu*/
22     const val=
23     await web3.eth.getBalance(props.currentAccount)
24     const bal=web3.utils.fromWei(val , "ether")
25     setBalance( bal )
26   }
27 }
28
```

—TokenConversion

Per la leggibilità dei dati all’utente, utilizziamo il pacchetto `web3.utils` e la funzione `web3.utils.fromWei()`, la quale richiede come parametri un valore (nel nostro caso il balance del token restituito dalla funzione `balanceOf()` a riga 8) e una stringa che definisce in quale formato trasformare questo valore (la stringa deve essere scelta seguendo un *unitMap* riportato in documentazione [19]). Verrà mostrato su schermo il valore “balance” settato tramite `useState` nella funzione `setTokenContract` in [TokenConversion](#).

Nel caso in cui venga selezionato Ethereum dal menu (il quale non è un token dello standard ERC-20 e non presenta un address specifico), il balance da mostrare riferito agli Ether disponibili nel wallet viene gestito dalla funzione a riga 22. La funzione in questione fa parte del pacchetto `web3.eth` e richiede come singolo parametro l’address dell’account da cui ottenere il dato; nel nostro caso è stato inserito l’account attualmente collegato all’applicazione.

Come si può notare in figura 3.16, il Card è composto, inoltre, da due form di input in cui è possibile selezionare il quantitativo di token che si vuole scambiare; una volta inseriti è possibile lanciare la chiamata allo smart-contract e alla relativa funzione per lo swap tramite il bottone con label “Swap”, la quale logica approfondiremo in [3.3.4](#).

Infine, si mostra la schermata generale comprensiva dei due Card appena descritti. La pagina mostrata in figura 3.17 riproduce la componente `Swap.js` con il suo path specifico settato nell’URL dell’applicazione. Come si evince dalla figura, il grafico visualizza la traccia esatta per i token selezionati dal Card riferito all’operazione di swap (nel nostro esempio LINK e DAI); è presente, inoltre, un bottoncino in alto a sinistra per cambiare la *currency* che l’utente desidera visualizzare nel grafico, il quale funzionamento è stato gestito tramite il meccanismo dei *Context*, che, in poche parole, ci permette di condividere dati considerati “globali” per un albero di componenti in React. Questo meccanismo può tornare utile per definire funzioni o variabili richiesti da molte componenti; siccome il cambio della valuta è un elemento

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

prioritario in un exchange, si è pensato di gestire il “cambio” in questa maniera.

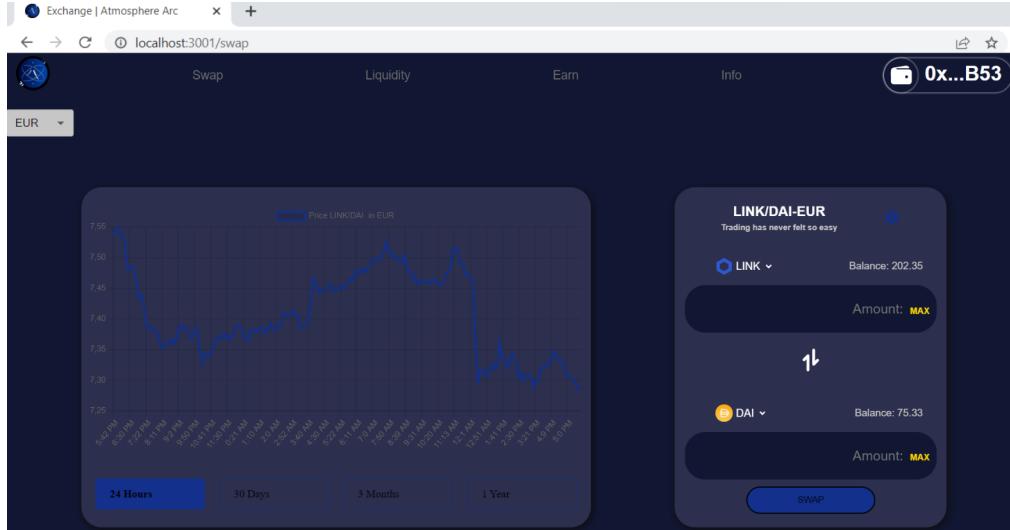


Figura 3.17: Schermata generale dello Swap.

Componente Liquidity.js

La componente **Liquidity.js** è la seconda parte del nostro exchange che si occupa della gestione e della creazione dei Pool di liquidità. Il Card in questione è molto simile a quello dello swap, con l'unica differenza che la logica dietro ad esso è molto differente (le differenze verranno preciseate nel dettaglio in [3.3.4](#)).

Per quanto riguarda l'aspetto grafico, come si vede in figura [3.18](#), le uniche due differenze con lo swap si denotano dal bottone che avvia la logica dell'operazione (con etichetta “Add Liquidity”) e nella visualizzazione dei valori inseriti dall'utente all'interno dei due form di input.

Il form è stato pensato in maniera tale che un utente, subito dopo l'inserimento di un dato quantitativo di token (e quindi dell'ipotetica liquidità da aggiungere all'interno della Pool), il sistema calcoli gli importi massimi di output per quel determinato valore, date le *riserve* per quel Pool.

L'operazione appena descritta è molto utile per visualizzare quanti token effettivi l'utente debba depositare nella Pool (ricordiamo che per depositare fondi in un Pool, è necessario fornire una stessa quantità di liquidità per token, come anticipato nel paragrafo [Pool di liquidità](#)).

Questo è reso possibile grazie ad una funzione scritta in **Liquidity.js**, mostrata di seguito:

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

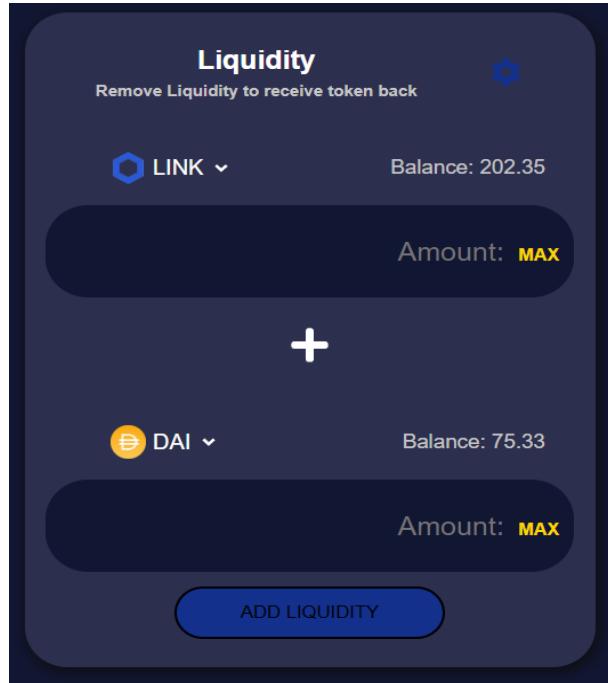


Figura 3.18: Interfaccia grafica Card per l'aggiunta di liquidità nei pool di liquidità.

```
1 const getPriceLiquidity = async (value) => {
2 const utilsSet = async (conversion) => {
3   if (PairAddress === NULLADDRESS) {
4     const valueConverted=convertToWei(conversion ,value)
5     setTokenA_Quantity(valueConverted)
6   }
7   else {
8     /*flag booleano (loading) per attendere che
9      l'operazione a riga 16 termini. Utilizzata per
10     visualizzare un loading animation durante l'esecuzione*/
11    setLoading(true)
12    const valueConverted=convertToWei(conversion ,value)
13    setTokenA_Quantity(valueConverted)
14    const path = [
15      tokenA.address ,
16      tokenB.address
17    ]
18    /*Chiamata alla funzione getAmountsOut() dallo
19      smart-contract del Router dell'exchange*/
20    const price =
21    await props.router.methods.
22    getAmountsOut(tokenA.Quantity , path).call()
23    const output = props.web3.utils.fromWei(price [1] , "ether")
```

```

24         setLoading(false)
25         setOutputQuantity(output)
26     }
27 }
28 const PairAddress=
29 await props.factory.methods.
30 getPair(tokenA.address, tokenB.address).call();
31 setPairAddress(PairAddress);
32 // Se i due token sono stati selezionati
33 if (tokenA && tokenB && value !== 0) {
34     if (tokenB.decimals === 18) {
35         await utilsSet("ether")
36     }
37     if (tokenB.decimals === 8) {
38         await utilsSet("gwei")
39     }
40     if (tokenB.decimals === 6) {
41         await utilsSet("mwei")
42     }
43 }
44 }
45

```

-getPriceLiquidity

Nella funzione `getPriceLiquidity()`, a seconda del tipo di token per il quale si deve calcolare l'importo massimo di output, viene lanciata la funzione `utilsSet()`, la quale, da riga 7, converte il valore del token A dato in input dall'utente nel formato del token B scelto per poi eseguire, alla riga 15 la funzione che ci permette di calcolare l'output atteso. La funzione in questione (`getAmountsOut()`) fa parte del *RouterContract* dell'exchange e richiede come parametri un valore (corrispondente al valore inserito dall'utente nel primo form di input) e un array di address, il quale deve contenere gli address dei token scelti dall'utente per i quali si vuole aggiungere liquidità (inizializzato, nel nostro caso, a riga 11). L'array in questione può essere personalizzato a proprio piacimento per applicare il concetto di [Routing](#).

3.3.4 Interazione con gli smart-contract

Come definito in [19], per l'interazione con gli smart-contract lato front-end ci viene in soccorso l'oggetto `web3.eth.Contract`. Questo oggetto (presente all'interno del pacchetto `web3.eth`) semplifica molto l'interazione con gli smart-contract sulla Blockchain di Ethereum. Al momento della creazione dell'oggetto `web3.eth.Contract`, bisogna passare come primo parametro l'interfaccia JSON (corrispondente all'ABI del contratto; la definizione è stata anticipata in [2.2.2](#)) del rispettivo contratto e, come secondo parametro, l'address dello stesso. Si mostra di seguito un esempio pratico.

```
new web3.eth.Contract(jsonInterface, address)
```

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

In questo paragrafo spiegheremo il modo con cui utilizzare l’oggetto appena citato e i vari metodi degli smart-contract da chiamare. Suddivideremo il paragrafo in diverse sezioni:

- Interazione con gli [ERC-20 Contracts](#);
- Interazione con il [Exchange Contracts](#) (operazioni per lo swap di token e per la gestione dei Pool di liquidità).

ERC-20 Contracts

Ricordiamo come lato front-end, una volta renderizzati i Card relativi allo swap e ai Pool di liquidità, venisse richiesta la selezione, da parte dell’utente, della coppia dei token. La selezione dei due token avviene attraverso le componenti [TokenA.js](#) e [TokenB.js](#), che renderizzano il menu (raffigurato in figura 3.16) e istanziano un oggetto `web3.eth.Contract` per ogni token selezionato dall’utente. La gestione dell’ERC-20 Contract è mostrata nello snippet di codice [TokenConversion](#), riportiamo la riga corrispondente dell’istanziamento:

```
new web3.eth.Contract(ERC20abi, token.address)
```

Siccome ERC-20 introduce uno standard per i token fungibili, al momento della creazione dello smart-contract riferito al token selezionato, passiamo come primo parametro l’ABI standardizzato (uguale per tutti), mentre, come secondo parametro inseriamo l’address del token scelto.

Come si può notare dal codice, l’address viene recuperato da un oggetto “token”, prelevato effettuando il map da un file JSON precedentemente compilato con la lista dei nomi di tutti i token, i decimals riferiti allo specifico token, l’address, il simbolo (l’acronimo del nome, ad esempio ETH per Ethereum) e un asset(l’immagine del token). Così facendo, rendiamo il codice usabile, facile da modificare con un’interfaccia grafica (il menu dei token mostrato in figura 3.16) intuitiva e molto semplice da utilizzare per l’utente finale.

Exchange Contracts

La maggior parte degli exchange decentralizzati basati sul meccanismo degli Automatic Market Maker (AMM), come la nostra applicazione, possiede, solitamente, due smart-contract principali:

- *Factory Contract*: sono presenti le funzioni che permettono la generazione dei Pool di liquidità. Ogni Pool è identificabile da un *Pair Contract* che definisce la coppia dei token. Nella nostra applicazione prende il nome di *AtmosFactory*;

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

- *Router Contract*: supporta le funzionalità di base svolte per interagire con i *Pair Contract*, fornisce una serie di funzioni di visualizzazione che verranno ampiamente utilizzate sul frontend del DEX e si interpone tra l'utente e il *Factory Contract*. Nella nostra applicazione prende il nome di *AtmosRouter*.

In questa parte dell'elaborato tratteremo la logica che c'è dietro alle operazioni che la nostra Dapp ci permette di fare, passando dalle operazioni di swap, fino all'aggiunta della liquidità per un Pool. Per quanto riguarda le operazioni di swap, nel nostro caso si sono diverse funzioni specifiche del *Router Contract*; la prima funziona implementata è `swapExactTokensForTokens()`, il codice, scritto in Solidity, viene mostrato di seguito:

```
1 function swapExactTokensForTokens(
2     uint256 amountIn, uint256 amountOutMin,
3     address[] calldata path, address to,
4     uint256 deadline
5 ) external virtual override
6 ensure(deadline) returns (uint256[] memory amounts) {
7     amounts =
8     AtmosLibrary.getAmountsOut(factory, amountIn, path);
9     require(amounts[amounts.length - 1] >= amountOutMin,
10    "AtmosRouter: INSUFFICIENT_OUTPUT_AMOUNT");
11
12     TransferHelper.safeTransferFrom(
13         path[0],
14         msg.sender,
15         AtmosLibrary.pairFor(factory, path[0], path[1]),
16         amounts[0]
17     );
18     _swap(amounts, path, to);
19 }
20
```

Come mostrato all'interno dello snippet, la funzione richiede cinque parametri distinti:

- *amountIn* corrispondente alla quantità di token da inviare;
 - *amountOut*, ossia la quantità minima di token di output che deve essere ricevuta;
 - Il *path*, ossia il “sentiero” che la funzione deve percorrere attraverso i pool di liquidità. Questi ultimi devono avere liquidità al loro interno;
 - Il *destinatario* dei token che verranno mandati;
 - La *deadline*, cioè la scadenza dopo la quale la transazione verrà ripristinata.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

L'*omountIn* corrisponde all'importo di token inserito nel form di input mostrato in figura 3.16, cioè il valore che l'utente decide di trasferire a seconda delle sue disponibilità. L'*amountOut* viene settato a zero per garantire il corretto funzionamento (non si è pensata a nessuna limitazione). Il *path* viene settato attraverso l'address dei due token scelti dall'utente nel Card dello swap.

Il *destinatario* dell'invio corrisponderà all'account in quel momento connesso alla Dapp, mentre la *deadline* è settata al momento del timestamp più dieci minuti di tolleranza nei quali l'utente deve completare e confermare la transazione. La precipita funzione scambia una quantità esatta di token (previa approvazione di un importo di indennità da dare al *Router*, vedere 3.6) di input con il maggior numero possibile di token di output, lungo il percorso determinato dal parametro *path*. Il primo elemento del percorso deve essere il token che si vuole scambiare, l'ultimo deve essere il token di output che si vuole ricevere e tutti gli elementi intermedi, invece, rappresentano coppie intermedie tramite cui scambiare (guardare il *Custom Path* di *Routing*).

A riga 8, attraverso la funzione *getAmountsOut()* (la spiegazione di questa funzione è stata anticipata in *getPriceLiquidity()*), si ricava l'*omount* calcolato dalle riserve della pool specifica per il nostro *path*; una volta fatto ciò si trasferiscono, attraverso la funzione *safeTransferFrom()*, i token del pool da un indirizzo all'altro. Essa richiede come parametri l'address del mittente, del ricevente, del Pair riferito alla coppia di token e l'*amount* definito precedentemente. Sarà poi la funzione *_swap()* a completare l'operazione in via definitiva. Si mostra lo snippet di codice presente all'interno della componente React *Swap.js* per l'interazione allo smart-contract del *Router*, la gestione della chiamata alla funzione *swapExactTokensForTokens()* e dei relativi parametri.

```
1 /* [...] */
2 const path = [
3   tokenA.address,
4   tokenB.address
5 ]
6 const deadline = Math.floor(Date.now() + (10 * 60))
7 const valueConverted = new BN(value)
8 setTokenA_Quantity(valueConverted)
9 try {
10 // Metodo del contratto AtmosRouter.sol per lo swap di token
11   await props.router.methods.swapExactTokensForTokens(
12     valueConverted,
13     new BN(0),
14     path,
15     props.currentAccount,
16     deadline
17   ).send({from: props.currentAccount, gas: 500000})
```

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

```

18     .then(tx => {
19         console.log("swapExactTokensForToken tx: ", tx)
20     });
21     } catch (error) {
22         console.log(error)
23     }
24 /* [...] */
25
                                         -swapExactTokensForTokens(front-end)

```

Ricordiamo come l'istanza del *Router Contract* venga settata al momento dell'`onLogin`, ogniqualvolta un nuovo utente accede all'applicazione. La chiamata riferita al contratto viene eseguita alla riga 11, utilizzando la funzione `send()` la quale, a differenza della funzione `call()` vista fin ora, altera lo stato dello smart-contract richiedendo un input all'utente per essere complete. La suddetta funzione permette di inserire dei parametri opzionali, ma molto importanti; nel nostro caso, come scritto a riga 17, si definiscono i parametri `from` e `gas`. Il parametro `from` è un address che rappresenta l'indirizzo da cui deve essere inviata la transazione, mentre il parametro `gas` corrisponde al gas limite previsto per questa transazione. I parametri della funzione `swapExactTokensForTokens` vengono inizializzati dalla riga 2 alla riga 8.

La seconda funzione implementata si tratta di `swapExactETHForTokens()`; di seguito viene mostrato il codice Solidity in questione:

```

1 function swapExactETHForTokens(
2     uint256 amountOutMin,
3     address[] calldata path,
4     address to,
5     uint256 deadline
6 ) external payable virtual override
7
8     ensure(deadline) returns (uint256[] memory amounts) {
9         require(path[0] == WETH, "AtmosRouter: INVALID_PATH");
10        amounts =
11            AtmosLibrary.getAmountsOut(factory, msg.value, path);
12        require(amounts[amounts.length - 1] >= amountOutMin,
13            "AtmosRouter: INSUFFICIENT_OUTPUT_AMOUNT");
14        IWETH(WETH).deposit{value: amounts[0]}();
15        assert(IWETH(WETH).transfer(AtmosLibrary
16            .pairFor(factory, path[0], path[1]), amounts[0]));
17        _swap(amounts, path, to);
18    }
                                         -swapExactETHForTokens

```

La logica di questa funzione è molto simile alla precedente, con l'unica differenza che si scambia una quantità esatta di ETH con il maggior numero possibile di token di output; quindi, se un utente lo desidera, ha la possibilità di utilizzare Ether per lo scambio. La funzione richiede quattro parametri:

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

- amountOutMin come per `swapExactTokensForTokens`;
- Il path corrispondente al percorso dei pool di liquidità, il primo elemento del percorso in questione deve necessariamente essere WETH⁵⁰ e l'ultimo il token di output;
- Il destinatario dei token che verranno mandati;
- La deadline come per `swapExactTokensForTokens`.

La funzione sopracitata ha un funzionamento molto simile alla funzione `swapExactTokensForTokens()`, con l'unica differenza che (dalla riga 14), viene richiesto il trasferimento di un numero equivalente all'amounts settato a riga 10 di WETH nella pool riferita al *Pair* della coppia WETH-tokenB, per poi concludere con la consueta chiamata a `_swap()` e completare l'operazione in via definitiva.

Per quanto riguarda l'interfacciamento smart-contract alla funzione appena descritta, la logica è praticamente la stessa di `swapExactTokensForTokens()` con la differenza che il *path* deve essere inizializzato con l'address dei WETH riferiti alla rete di test Ropsten (nel nostro caso specifico). Di seguito mostriamo il codice presente all'interno della componente `Swap.js`:

```
1 /* [...] */
2 const WEIH = await props.router.methods.WEIH().call()
3 const path = [
4   WETH,
5   tokenB.address
6 ]
7 const deadline = Math.floor(Date.now() + (10 * 60))
8 const valueConverted = new BN(value)
9 setTokenA_Quantity(valueConverted)
10 try {
11   //Metodo del contratto AtmosRouter.sol per lo swap di token
12   await props.router.methods.swapExactETHForTokens(
13     valueConverted,
14     new BN(0),
15     path,
16     props.currentAccount,
17     deadline
18   ).send({from: props.currentAccount, gas: 500000})
19   .then(tx => {
20     console.log(`swapExactETHForTokens tx: ${tx}`)
21   });
22 } catch (error) {
```

⁵⁰WETH (o Wrapped Ether) è un tipo speciale di token ERC-20 il cui scopo è facilitare diverse operazioni di scambio nell'ecosistema Ethereum(principalmente scambi di Ether per altri gettoni ERC-20).

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

```

23         console.log(error)
24     }
25     /*[...]*/
26
    -swapExactETHForTokens(front-end)

```

Come si può notare alla riga 2 viene prelevato l'address del token WETH attraverso una chiamata dello smart-contract del *Router*: **WETH()**. Il valore risultante viene inserito nella prima posizione del percorso, nella variabile **path**, come anticipato in precedenza. Per il resto, l'assegnazione, l'invio della transazione e la chiamata della funzione restano identici.

Da precisare come queste non siano le uniche due funzioni (per le operazioni di swap) presenti all'interno dello smart-contract del *Router*, bensì ne sono presenti diverse per ogni necessità. La logica di queste altre funzioni restano molto simili, con l'unica differenza che l'ordine con cui i percorsi vengono assegnati cambia, esempi di funzioni sono: **swapTokensForExactTokens()** (il quale riceve una quantità esatta di token di output e non di input), **swapExactTokensForETH()** (il quale permette di *ricevere* ETH invece di mandarli).

Una volta discusso delle funzioni riferite allo swap dei token, parliamo di una delle funzioni più importanti di un DEX, ossia quella per l'**_addLiquidity()**. La seguente funzione viene mostrata di seguito:

```

1 function _addLiquidity(
2     address tokenA, address tokenB, uint256 amountADesired,
3     uint256 amountBDesired, uint256 amountAMin, uint256
4     amountBMin
5     ) internal virtual returns
6     (uint256 amountA, uint256 amountB) {
7     /* Creazione del Pair Contract se non esiste */
8     if (IAtmosFactory(factory)
9         .getPair(tokenA, tokenB) == address(0)) {
10         IAtmosFactory(factory).createPair(tokenA, tokenB);
11     }
12
13     /*Se si e' primi ad aggiungere liquidita'
14      si determinera' il prezzo iniziale del Pool*/
15     (uint256 reserveA, uint256 reserveB) =
16     AtmosLibrary.getReserves(factory, tokenA, tokenB);
17
18     if (reserveA == 0 && reserveB == 0) {
19         (amountA, amountB) =
20         (amountADesired, amountBDesired);
21     }
22     else {
23         uint256 amountBOptimal =
24         AtmosLibrary.quote(amountADesired, reserveA, reserveB);
25         if (amountBOptimal <= amountBDesired) {

```

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

```

25     require(amountBOptimal >= amountBMin,
26     "AtmosRouter: INSUFFICIENT_B_AMOUNT") ;
27     (amountA , amountB) =
28     (amountADesired , amountBOptimal) ;
29   } else {
30     uint256 amountAOptimal =
31     AtmosLibrary . quote(amountBDesired , reserveB , reserveA ) ;
32     assert (amountAOptimal <= amountADesired) ;
33     require(amountAOptimal >= amountAMin ,
34     "AtmosRouter: INSUFFICIENT_A_AMOUNT") ;
35     (amountA , amountB) =
36     (amountAOptimal , amountBDesired) ;
37     }
38   }
39 }
```

—_addLiquidity

La seguente funzione ha bisogno di otto parametri, definiti di seguito:

- L'address del *tokenA*;
- L'address del *tokenB*;
- *amountADesired*: la quantità di tokenA da aggiungere come liquidità;
- *amountBDesired*: la quantità di tokenB da aggiungere come liquidità;
- *amountAMin*: limita il fatto che il prezzo del tokenB/ tokenA possa aumentare prima che la transazione avvenga;
- *amountBMin*: limita il fatto che il prezzo del tokenA/ tokenB possa aumentare prima che la transazione avvenga.

Mentre, come valori di ritorno, restituisce la coppia (*amountA,amountB*). A riga 7, la funzione parte con un controllo chiamando la funzione *getPair()* del *Factory Contract*. Questa funzione serve per ottenere l'indirizzo del *Pair Contract* dati i due token scelti dal front-end. Tramite questa funzione, si verifica se è uguale ad un address nullo; se sì, viene creato il *Pair Contract* corrispondente alla coppia tramite la funzione dello smart-contract della *Factory* chiamata *createPair()*.

Subito dopo aver effettuato questo controllo preliminare, si verifica, a riga 17, se le riserve dei due token sono uguali a zero (praticamente se si è i primi a voler aggiungere liquidità) allora si aggiunge la liquidità con i valori passati in *amountADesired* e *amountBDesired*, determinando, di conseguenza, il prezzo iniziale della Pool, in quanto primi Liquidity Provider (LP).

Se questo non fosse il caso, da riga 21 in poi si setta la variabile *amountBOptimal* con la funzione *quote()* dello smart-contract della *Library*. Essa richiede tre parametri specifici: come primo parametro un importo di asset

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

(in questo caso **amountADesired**) e come secondo e terzo parametro le riserve dei due token; la funzione attraverso dei calcoli matematici (mostrati di seguito):

$$\text{amountBOptimal} = (\text{amountADesired} \cdot \text{reserveB}) / \text{reserveA}$$

trova l'importo ottimale del token B.

Una volta fatto ciò, a riga 24, si controlla se il valore trovato è minore o uguale della quantità di token che si vogliono aggiungere alla liquidità (**amountBDesired**), se va a buon fine, si richiede, alla riga successiva, che tale valore sia anche maggiore o uguale a **amountBMin** (per evitare che l'operazione non sia più conveniente) e solo se anche questa condizione è verificata verrà restituita la coppia (**amountADesired,amountBOptimal**).

Se, invece, la prima condizione citata a riga 24, non venisse verificata, si riesegue lo stesso identico codice, ma per il token A.

Per quanto riguarda la chiamata dell'aggiunta della liquidità da parte del front-end (appena l'utente ha inserito tutti i dati e ha interagito con il bottone, come mostrato in [Liquidity.js](#)) si utilizza un'altra funzione sempre facente parte del *Router Contract* ossia [addLiquidity\(\)](#):

```
1 function addLiquidity(
2     address tokenA, address tokenB,
3     uint256 amountADesired, uint256 amountBDesired,
4     uint256 amountAMin, uint256 amountBMin, address to,
5     uint256 deadline
6 ) external virtual override ensure(deadline)
7 returns (
8     uint256 amountA,
9     uint256 amountB,
10    uint256 liquidity
11 )
12 {
13     (amountA, amountB) =
14     _addLiquidity(tokenA, tokenB,
15     amountADesired, amountBDesired,
16     amountAMin, amountBMin);
17     address pair = AtmosLibrary
18     .pairFor(factory, tokenA, tokenB);
19     TransferHelper
20     .safeTransferFrom(tokenA, msg.sender, pair, amountA);
21     TransferHelper
22     .safeTransferFrom(tokenB, msg.sender, pair, amountB);
23     liquidity = IAtmosPair(pair).mint(to);
24 }
```

Questa è la funzione che lanceremo dal front-end, la quale possiede gli stessi identici parametri della funzione [_addLiquidity\(\)](#), con l'aggiunta dei parametri *deadline* e *to* (riferito all'address del destinatario). Partendo da

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

riga 13, la funzione richiama `_addLiquidity()` coi relativi parametri aggiunti, per farsi restituire la coppia `(amountA,amountB)` i quali valori verranno utilizzati poi dopo per il trasferimento

A riga 17 viene inizializzata una variabile di tipo address tramite la funzione `pairFor()` del *Library Contract*, la quale non fa altro che calcolare l'indirizzo per una coppia senza effettuare chiamate esterne alla *Factory*, per l'appunto, si passa come parametro proprio l'address della *Factory* e dei due token. La variabile `factory` in questo contesto è una *variabile globale*⁵¹ dello smart-contract del *Router*.

Successivamente, a riga 19 e 21, chiamiamo due volte la funzione `safeTransferHelper()` che richiede come parametri quattro valori:

- L'address del token da mandare al *Pair*;
- L'address del `msg.sender`, ossia colui che lancia la funzione (nel nostro caso l'utente attualmente collegato alla Dapp);
- L'address del *Pair* calcolato prima;
- L'amount (il quantitativo) del rispettivo token;

La funzione in questione viene chiamata due volte perché, come ricordiamo, quando si deve aggiungere liquidità (come anticipato in [Pool di liquidità](#)), è necessario fornire la stessa quantità di liquidità per token. Proprio per questo si utilizza in `_addLiquidity()` la funzione `quote()` per il calcolo del valore preciso da assegnare ai due token. La funzione `safeTransferHelper()`, quindi, permette di far mandare un quantitativo preciso di token al contratto *Pair* da parte dell'utente chiamante.

Una volta descritte le funzioni utilizzate nel linguaggio Solidity, analizziamo come ci si è comportati, lato front-end per la chiamata della funzione per l'aggiunta della liquidità.

Di seguito si mostra lo snippet di codice di riferimento:

```
1 /* [...] */  
2 const addLiquidity = async (value) => {  
3 /*Se i due token esistono e se il valore  
4 inserito nel form e' diverso da zero*/  
5 if(tokenA && tokenB && value!==0) {  
6   const tokenAContract =  
7     SetERC20Contract(props.web3,tokenA.address)  
8   const tokenBContract =  
9     SetERC20Contract(props.web3,tokenB.address)  
10  await ApproveRouter(props.web3,props.currentAccount,  
11    tokenAContract,ROUTER_ADDRESS)
```

⁵¹variabili che possono venire usate da tutte le funzioni e dal programma principale.

CAPITOLO 3. IMPLEMENTAZIONE DEGLI APPLICATIVI

```
12  await ApproveRouter(props.web3, props.currentAccount,
13    tokenBContract,ROUTER_ADDRESS)
14  await props.router.methods.addLiquidity(
15    tokenA.address,
16    tokenB.address,
17    tokenA_Quantity,
18    outputQuantity,
19    0,
20    0,
21    props.currentAccount,
22    Math.floor(Date.now() / 1000) + 60000 * 10
23  ).send({from:props.currentAccount,gas:500000})
24  .then(tx => console.log("\addLiquidity tx: "+tx))
25  }
26  }
27 /*[...]*/
28
29 /*Funzione chiamata a riga 10 e 12*/
30 export async function ApproveRouterSwap
31 (web3,currentAccount,_contract,router) {
32   const value=web3.utils
33   .toBN("11579208923731619542357098500868790785326
34   9984665640564039457584007913129639935");
35
36   await _contract.methods
37     .approve(router, value)
38     .send({from: currentAccount})
39 }
40
  -addLiquidity(front-end)
```

La funzione in questione viene chiamata all'onClick del bottone con etichetta “addLiquidity” visualizzato in figura 3.18. Prima di poter chiamare la funzione dello smart-contract del nostro *Router*, bisogna inizializzare gli oggetti `web3.eth.Contract` dei due token; a rigga 7 e 9 vengono istanziati attraverso la funzione `SetERC20Contract()` per evitare ridondanze.

Gli oggetti istanziati ci serviranno per lanciare la funzione `ApproveRouter()`, la quale lancia l’`approve()` del contratto del token passato come parametro. Un caso d’uso comune è quello di approvare i token attraverso la metodologia utilizzata in questa funzione, infatti, come si vede a riga 33, si setta una variabile con il massimo intero possibile da poter utilizzare in Solidity. Così facendo il *Router Contract* può trasferire tutti i token per evitare all’utente procedure continue di approvazione, rallentando le operazioni e le procedure dei servizi offerti dal DEX. Ovviamente, prima di accettare l’approvazione di tale ammontare, l’utente deve essere consapevole dell’affidabilità della piattaforma su cui operare. Un esempio di piattaforma che utilizza questa metodologia è proprio Uniswap [26].

Capitolo 4

Conclusioni e sviluppi futuri

La sfida principale che si prepara ad affrontare la DeFi (e in generale il mondo Web3.0) è quella di diventare globale e accessibile a tutti. L'obiettivo di questo elaborato di tesi è proprio quello di realizzare una piattaforma moderna, che adopera framework e librerie innovative per la creazione di applicazioni decentralizzate, in grado di migliorare l'usabilità e la scalabilità dei servizi offerti solitamente da un'applicazione DeFi.

Per garantire che tutti gli utenti possano beneficiare dei vantaggi della finanza decentralizzata, è necessario educare le persone attraverso piattaforme di facile utilizzo. L'obiettivo è quello di offrire strumenti capaci di rendere più facile familiarizzare con la blockchain e di garantire una vera decentralizzazione con una conseguente padronanza, completa, delle proprie finanze senza avere l'obbligo e la necessità di affidarsi ad enti centrali.

Possibili sviluppi futuri possono essere diversi; sicuramente sarà importante garantire certezze all'utente attraverso la comunicazione di progetti sicuri, regolamentati in maniera chiara e trasparente. Tutto ciò aiuterebbe molto il suo riconoscimento a livello globale come vera alternativa alla finanza tradizionale. Il recente interesse che si sta avendo nel mondo delle criptovalute è sicuramente un passo avanti del processo.

In conclusione, ricordiamo, che l'elaborato esposto, oltre ad aver mostrato esempi di interfacce utente, ha avuto come primario obiettivo quello di sviluppare un exchange decentralizzato basato su smart-contract. Il lavoro di tesi in questione ha dimostrato che l'applicazione sviluppata permette agli utenti di operare su un vero sistema AMM. Allo stesso tempo, viene anche confermato come un'applicazione di questo tipo possa essere intuitiva e facile da utilizzare, capace di comunicare in maniera semplice con la blockchain e di rendere l'esperienza utente leggera e molto gradevole.

Bibliografia

- [1] Cryptography Definition from Wikipedia. URL: <https://it.wikipedia.org/wiki/Crittografia>.
- [2] Tanenbaum et al. *"Sistemi distribuiti. Principi e paradigmi"*. 2007.
- [3] Bitcoin Definition from Wikipedia. URL: <https://it.wikipedia.org/wiki/Bitcoin>.
- [4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Agosto 2008. URL: <https://bitcoin.org/bitcoin.pdf>.
- [5] Consensus Algorithm Definition from Binance Academy. URL: <https://academy.binance.com/it/articles/what-is-a-blockchain-consensus-algorithm>.
- [6] The Merge: the Ethereum Mainnet Merging. URL: <https://ethereum.org/en/upgrades/merge/>.
- [7] Cesar Castellon, Swapnoneel Roy, Patrick Kreidl, Ayan Dutta, and La-dislau Bölöni. Energy efficient merkle trees for blockchains. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1093–1099, 2021.
- [8] Ethereum Definition from Wikipedia. URL: <https://ethereum.org/it/what-is-ethereum/>.
- [9] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. 2014. URL: <https://ethereum.org/en/whitepaper/>.
- [10] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkeley, CA, 2017.
- [11] Fabian Schär. Decentralized finance: On blockchain- and smart contract-based financial markets. 2021.

-
- [12] NFT Definition from Investopedia. URL: <https://www.investopedia.com/non-fungible-tokens-nft-5115211>.
 - [13] Industry 4.0 Definition from Wikipedia. URL: https://it.wikipedia.org/wiki/Industria_4.0.
 - [14] Solidity Documentation. URL: <https://docs.soliditylang.org/en/v0.8.17/>.
 - [15] Block Explorer Definition from Binance Academy. URL: <https://academy.binance.com/en/glossary/block-explorer>.
 - [16] Truffle: The Development Framework for Ethereum. URL: <https://trufflesuite.com/truffle/>.
 - [17] Truffle Migration from Truffle Documentation. URL: <https://trufflesuite.com/docs/truffle/getting-started/running-migrations/#network-considerations>.
 - [18] React: A JavaScript library for building user interfaces. URL: <https://reactjs.org/>.
 - [19] Web3.js Library Documentation. URL: <https://web3js.readthedocs.io/en/v1.8.0/#>.
 - [20] Ethereum JSON-RPC Specification: a specification of the standard interface for Ethereum clients. URL: <https://playground.open-rpc.org/>.
 - [21] Gerardo Iovane: “Atmosphere Arc The first Constellation into the Blockchain”. URL: <https://www.atmospherearc.com/atmospherearc.pdf>.
 - [22] Uniswap Docs from Uniswap Protocol. URL: <https://docs.uniswap.org/>.
 - [23] Uniswap Pool Concept. URL: <https://docs.uniswap.org/protocol/V2/concepts/core-concepts/pools>.
 - [24] Uniswap Swap Concept. URL: <https://docs.uniswap.org/protocol/V2/concepts/core-concepts/swaps>.
 - [25] CoinGecko API Documentation. URL: <https://www.coingecko.com/en/api/documentation>.
 - [26] Uniswap Protocol. URL: <https://uniswap.org/>.

Elenco delle figure

1.1	Struttura di un blocco della catena	4
1.2	Schema base di una catena di blocchi	5
1.3	Esempio di tabella hash	6
1.4	Transaction Chain	9
1.5	Merkle Tree Root	10
1.6	Grafico del tasso inflattivo e della supply di Bitcoin	12
1.7	Struttura di un'applicazione DeFi.	16
2.1	Rappresentazione grafica del processo di lettura/scrittura delle funzioni di uno smart-contract	20
2.2	File di configurazione truffle-config.js	21
3.1	Sidebar dell'applicativo riferito allo staking	27
3.2	Footer dell'applicativo riferito allo staking	28
3.3	Albero delle componenti dell'applicativo riferito allo staking .	28
3.4	Main Card per il balance e le revenue	29
3.5	Card con i form per la messa in stake e in lock dei token . .	29
3.6	Schermata di approvazione dei token ATMOS	30
3.7	Schermata di warning per la mancata approvazione dei token	31
3.8	Rappresentazione grafica di un Pool di Liquidità	33
3.9	Rappresentazione grafica del funzionamento per il deposito liquidità da parte dei Liquidity Provider (LP)	34
3.10	Rappresentazione grafica del funzionamento relativo al trading di token	35
3.11	Custom Path per il Routing tra pool di liquidità	36
3.12	Albero delle componenti dell'applicativo riferito all'exchange	37
3.13	Scelta provider per la connessione alla Dapp	39
3.14	Schermata di caricamento per l'inserimento dei dati di accesso	41
3.15	Line Chart implementato nell'applicativo riferito all'exchange	43
3.16	Selezione dei token per il Card relativo allo Swap	44
3.17	Schermata generale dello Swap	46
3.18	Card per l'aggiunta di liquidità nei pool di liquidità	47

Elenco dei codici

3.1	Navbar Code (staking)	26
3.2	App.js (exchange)	38
3.3	onLogin() Function	39
3.4	API Code	42
3.5	Token Conversion Code	44
3.6	getPriceLiquidity() Function	47
3.7	swapExactTokensForTokens() Function	50
3.8	Call swapExactTokensForTokens Function (front-end)	51
3.9	swapExactETHForTokens() Function	52
3.10	Call swapExactETHForTokens() Function (front-end)	53
3.11	_addLiquidity() Function	54
3.12	addLiquidity() Function	56
3.13	Call addLiquidity() Function (front-end)	57