Manuale Sviluppatore
Progetto ShopChain

yakuzaishi.swe@gmail.com

Informazioni sul documento

Responsabile	Luca Carturan
Redattori	Michele Filosofo, Francesco Mattarello
Verificatori	Luca Busacca, Matteo Midena
Uso	Esterno
Destinatari	Prof. Tullio Vardanega Prof. Riccardo Cardin Ing. Fabio Pallaro - Sync Lab S.r.l.
Versione	1.0.0

Sommario

Questo documento racchiude tutte le informazioni necessarie per l'estensione e la manutenzione del software ShopChain.

Registro delle Modifiche

Versione	Data	Autore	Ruolo	Descrizione
1.0.0	2022/06/09	Luca Carturan	Responsabile	Approvato per il rilascio
0.3.0	2022/06/08	Matteo Midena	Verificatore	Verifica generale del documento.
0.2.2	2022/06/07	Michele Filosofo, Luca Busacca	Programmatore, Verificatore	Stesura §9 e verifica.
0.2.1	2022/06/04	Francesco Mattarello, Matteo Midena	Programmatore, Verificatore	Stesura §7 e §8 e verifica.
0.2.0	2022/06/02	Luca Busacca	Verificatore	Verifica generale del documento.
0.1.3	2022/06/01	Francesco Mattarello, Matteo Midena	Programmatore, Verificatore	Terminata stesura §6 e verifica.
0.1.2	2022/05/31	Michele Filosofo, Luca Busacca	Programmatore, Verificatore	Stesura §5, inizio stesura §6 e verifica.
0.1.1	2022/05/29	Francesco Mattarello, Matteo Midena	Programmatore, Verificatore	Stesura §4 e verifica.
0.1.0	2022/05/28	Matteo Midena	Verificatore	Verifica generale del documento.
0.0.4	2022/05/27	Michele Filosofo, Luca Busacca	Programmatore, Verificatore	Terminata stesura §3 e verifica.
0.0.3	2022/05/25	Michele Filosofo, Matteo Midena	Programmatore, Verificatore	Stesura §2, inizio stesura §3 e verifica.
0.0.2	2022/05/23	Francesco Mattarello, Luca Busacca	Programmatore, Verificatore	Stesura §1 e verifica.
0.0.1	2022/05/21	Francesco Mattarello, Matteo Midena	Programmatore, Verificatore	Creata struttura documento e verifica.

Contenuti

Elenco delle tabelle

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

Il fine di questo documento è quello di creare una linea guida per gli sviluppatori che andranno ad estendere o a manutenere il prodotto *ShopChain*. Di seguito lo sviluppatore troverà nel documento tutte le informazioni riguardanti i linguaggi e le tecnologie utilizzate, l'architettura del sistema e le scelte progettuali effettuate per il prodotto. Questo documento ha anche il fine di illustrare la procedura di avvio dell' applicazione.

1.2 Scopo del prodotto

L'avvento delle tecnologie blockchain$_G$ ha portato e porterà nei prossimi anni a grandi cambiamenti nella società. In particolare, ha aperto le porte a una nuova forma di finanza, la cosiddetta "DeFi" (Finanza Decentralizzata) che ha permesso a chiunque sia dotato di connessione internet di creare un wallet$_G$ e possedere quindi criptovalute$_G$. Questo ha delineato due profili critici strettamente legati; da un lato il controllo del proprio portafoglio è passato completamente nelle mani dell'utente, dall'altro ha comportato la mancanza di un ente terzo che si occupi di gestire transazioni e offrire garanzie.
Lo scopo di *ShopChain* è quindi il seguente: creare un e-commerce$_G$ basato su blockchain$_G$, dove il pagamento avviene tramite cryptovalute$_G$, tutelando le parti coinvolte nell'acquisto.
Il risultato finale è la realizzazione di un prototipo di una piattaforma integrabile con un "crypto-ecommerce$_G$", che si occupi di gestire gli ordini dalla fase di pagamento fino alla consegna del prodotto.

1.3 Glossario

Per evitare ambiguità relative alle terminologie utilizzate, queste verranno evidenziati da una 'G' a pedice e riportate nel glossario alla fine del documento.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- Capitolato d'appalto C2 - ShopChain: Exchange platform on blockchain.

  https://www.math.unipd.it/~tullio/IS-1/2021/Progetto/C2.pdf

1.4.2 Riferimenti informativi

- Slide corso di Ingegneria del Software, in particolare:
  - Slide P2 - Diagrammi delle classi e dei package

    https://www.math.unipd.it/~rcardin/swea/2021/Diagrammi%20delle%20Classi_4x4.pdf
  - Slide P5 - Diagrammi di sequenza

    https://www.math.unipd.it/~rcardin/swea/2022/Diagrammi%20di%20Sequenza.pdf
  - Slide L03 - Principi SOLID

    https://www.math.unipd.it/~rcardin/sweb/2022/L03.pdf
  - Slide L01 - Design pattern comportamentali

    https://www.math.unipd.it/~rcardin/sweb/2022/L01.pdf
  - Slide L02 - Design pattern architetturali: Model View Controller e derivati

https://www.math.unipd.it/~rcardin/sweb/2022/L02.pdf

- Libreria per l'implementazione dell' observer pattern:

  https://mobx.js.org/README.html

- Documentazione Solidity:

  https://docs.soliditylang.org/en/v0.8.14/

2 Tecnologie coinvolte

Tecnologia	Versione	Descrizione
Linguaggi		
Typescript	4.6.3	Linguaggio fortemente tipato, basato su Javascript, più scalabile
Solidity	1.0	Linguaggio usato per la creazione di Smartcontracts su varie blockchain, in particolare tutte quelle basate su Ethereum
HTML	5	Utilizzato nel progetto assieme a React per definire l'interfaccia grafica
Sass	1.47.0	Estensione di CSS, utilizzato per definire la formattazione dei documenti HTML5 e lo stile
Strumenti		
Npm	7.x	Gestore di pacchetti utilizzato per effettuare le operazioni di build del codice
Babel	7.9.6	Transcompiler JavaScript utilizzato per convertire il codice ECMAScript 2015+ in una versione retro compatibile per browser non aggiornati
Truffle	5.4.33	Ambiente di test locale per Smartcontract
Docker	4.9.0	Strumento open-source che automatizza il processo di deployment di applicazioni all'interno di contenitori software
Librerie		
React	16.3.1	Libreria per la creazione di UI scelta per facilitare lo sviluppo del front-end e avere performance migliori grazie al suo metodo di renderizzazione dei componenti grafici
MobX	6.5	Libreria per React che permette la gestione dello stateG dei componenti e l'implementazione del design pattern Observer
Web3.js	1.7.0	Collezione di librerie per interagire con nodi ethereum locali o remoti

Tabella 2: Tabella delle tecnologie

3 Setup

In questa sezione vengono trattati i requisiti minimi necessari per l'utilizzo dell'applicazione ShopChain e successivamente come poter installare il prodotto in locale direttamente dal repository$_G$ pubblico GitHub$_G$ del gruppo Yakuzaishi, accessibile al seguente link: link al repository.

3.1 Requisiti di sistema

Per far si che le operazioni di installazione e avvio del prodotto avvengano correttamente e che si possa aver accesso a tutte le funzionalità, è necessario avere installati nella propria macchina i seguenti software.

Software	Versione	Link al download
Docker	4.9.0	https://www.docker.com/products/docker-desktop/

Tabella 3: Requisiti di sistema

Grazie all'utilizzo di Docker$_G$, abbiamo facilitato la procedura di setup dell'applicazione, in quanto Docker stesso si occupa di inizializzare tutte le tecnologie necessarie alla compilazione.

3.2 Requisiti hardware

Per avere delle prestazioni accettabili dell'applicazione è preferibile possedere almeno i seguenti componenti hardware.

Componente	Requisito
RAM	4GB DDR4
CPU	Processore 64-bit con Second Level Address Translation (SLAT)
SSD (o HDD)	3GB

Tabella 4: Requisiti hardware

Attenzione! In alcune CPU la virtualizzazione utilizzata da Docker per avviarsi non è abilitata di default, per abilitarla si deve accedere al BIOS$_G$ di sistema, e attivare l'impostazione correlata. Essendo gli ambienti BIOS molto variabili da macchina a macchina, non ci è possibile dare una indicazione precisa su dove trovare l'impostazione da cambiare, al seguente link troverete una guida generale di esempio: CPU virtualization.

È necessaria inoltre una connessione internet stabile per garantire un servizio ottimale, senza vincoli di banda.

3.3 Browser

L'applicazione è stata testata e quindi resa compatibile con le ultime versioni dei browser che supportano l'estensione Metamask$_G$, il cui setup verrà discusso nella sezione §3.4.

Componente	Requisito
Google Chrome	98
Microsoft Edge	98
Mozilla Firefox	97
Brave Browser	1.36

Tabella 5: Requisiti browser

3.4 Metamask

L'applicazione fa uso del plugin Metamask$_G$per la gestione dei wallet e dei pagamenti, al seguente link è possibile trovare la pagina ufficiale di download:

`https://MetaMask.io/download/`

Alla pagina sarà presente un bottone con scritto `"Install MetaMask"`

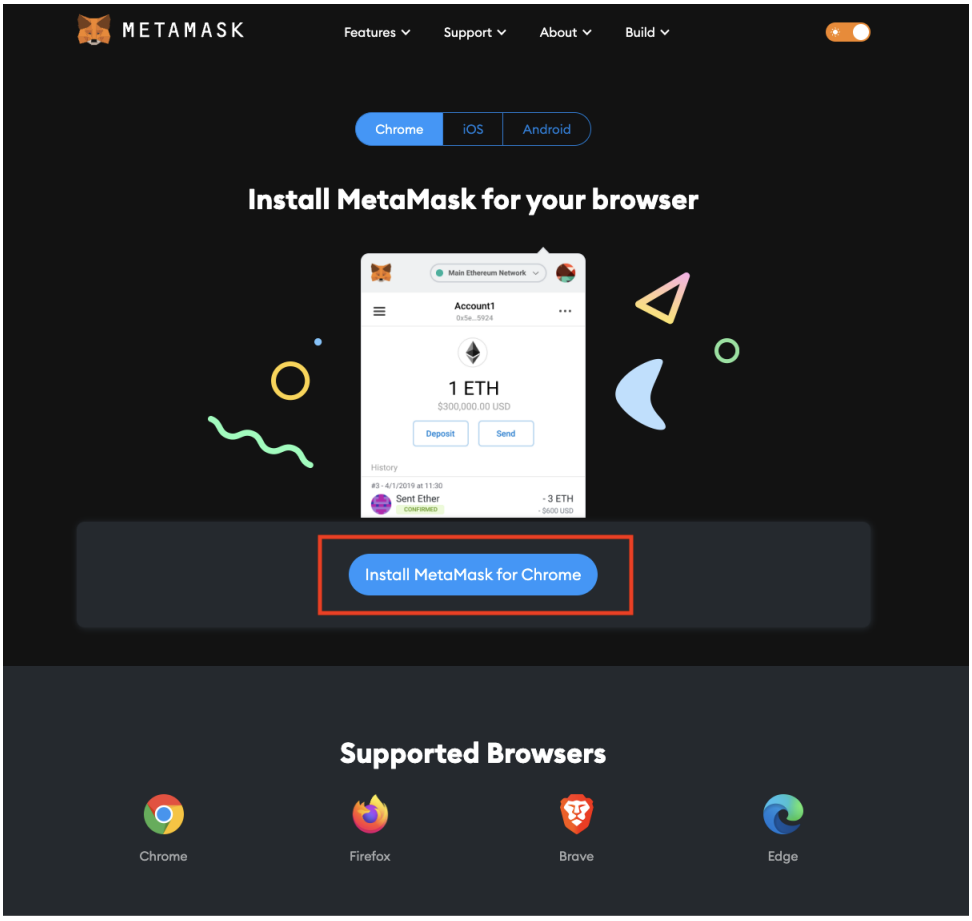

Figura 1: Pagina download MetaMask

Cliccando il bottone si verrà reindirizzati alla seguente pagina web nella quale basterà cliccare il bottone *"Aggiungi"*:

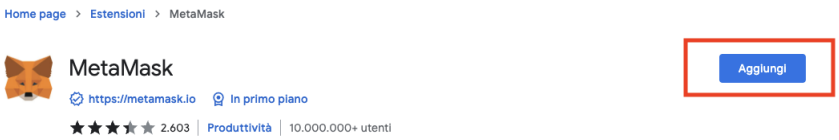

Figura 2: Aggiungere l'estensione MetaMask al browser (1)

Si aprirà dunque il popup mostrato in seguito nel quale sarà sufficiente cliccare `"Aggiungi estensione"`

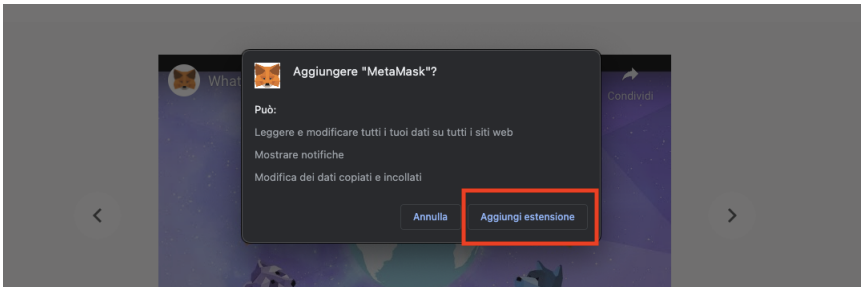

Figura 3: Aggiungere l'estensione MetaMask al browser (2)

Una volta che l'estensione sarà stata aggiunta si verrà automaticamente reindirizzati alla pagina mostrata nella figura seguente nella quale sarà sufficiente cliccare *Inizia*

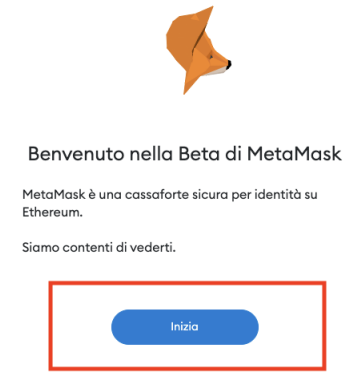

Figura 4: Pagina iniziale MetaMask

A questo punto non resta che scegliere se iniziare creando un nuovo portafoglio o se caricarne uno già esistente.

Figura 5: Scelta iniziale MetaMask: crea/importa portafogli

dopo aver scelto basterà seguire le semplici istruzioni riportate da MetaMask per completare il collegamento.

3.4.1 Configurazione

Per utilizzare l'applicazione ShopChain è necessario disporre di un account MetaMask impostato sulla testnet Fantom. Per aggiungere una nuova rete al proprio account MetaMask, cliccare sull'estensione MetaMask, quindi su `"Rete Ethereum Principale"` e infine `"Aggiungi Rete"` come mostrato di seguito:

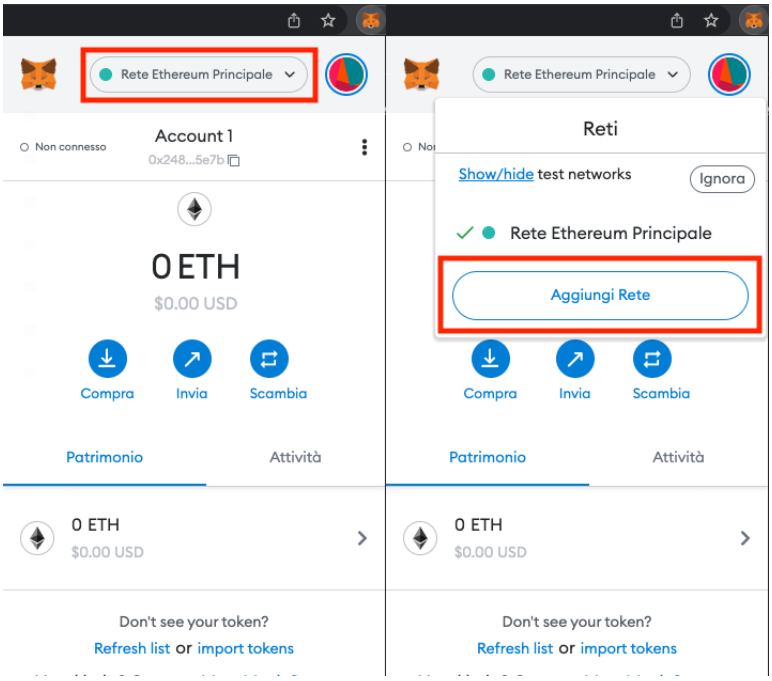

Figura 6: Inizio configurazione Fantom Testnet

Si verrà reindirizzati a una pagina contenente un form da riempire come mostrato di seguito:

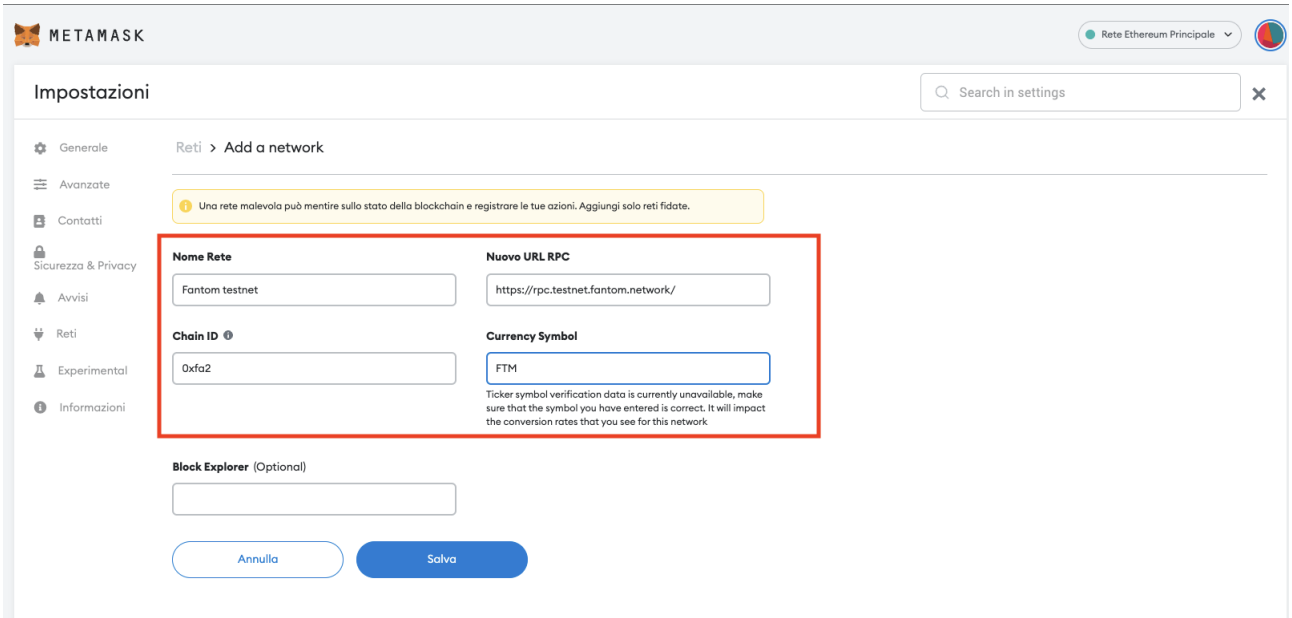

Figura 7: Form per l'aggiunta della rete Fantom Testnet

Per comodità vengono riportati i dati:

- Network Name: Fantom testnet;
- New RPC Url: https://rpc.testnet.fantom.network/;
- ChainID: 0xfa2
- Symbol: FTM

Una volta riempito il form cliccare il tasto `Salva`.
A questo punto, una volta caricato denaro nel portafogli tramite la Fantom faucet, il setup sarà completo.

Attenzione! Si raccomanda inoltre di notare che ciascuna operazione che prevede l'interazione con MetaMask (quali il pagamento, lo sblocco, il rimborso, la creazione della moneybox e il versamento di un contributo) prevedono il pagamento di gasfee, ovvero un ammontare variabile, generalmente piccolo, come commissione per il pagamento.
Tale ammontare è indipendente da ShopChain o MetaMask, risulta invece essere indispensabile per permettere alla blockchain di registrare i blocchi all'interno dei quali è possibile reperire le transazioni.

4 Inizializzazione

Per utilizzare l'applicazione web è necessario:

- Clonare il repository;
- Installare le dipendenze;
- Avviare la web app.

4.1 Clonare il repository

1. Scaricare il codice come file .zip direttamente dal repository shopchain-frontend:

   https://github.com/Yakuzaishi-SWE/shopchain-frontend

   Oppure, con Git$_{G}$installato in locale, è possibile clonare il repository con il comando:

   ```
   git clone https://github.com/Yakuzaishi-SWE/shopchain-frontend
   ```

2. Se non clonato tramite Git, estrarre il file .zip.

4.2 Installare le dipendenze

1. Spostarsi all'interno del repository clonato tramite terminale con il comando:

   ```
   cd percorso/shopchain-frontend
   ```

2. Eseguire il comando:

   ```
   npm i
   ```

3. Attendere il termine della procedura di installazione delle dipendenze.

4.3 Avviare la web app

1. Dalla cartella, eseguire il comando:

   ```
   npm start
   ```

2. il browser di default si aprirà in automatico con l'applicativo eseguito.

5 Strumenti di analisi e integrazione del codice

Strumento	Versione	Descrizione
Analisi Statica		
ESLint	8.9.0	Strumento di analisi statica utilizzato per la segnalazione degli errori di sintassi, per avere regole d'indentazione uguali in tutti i file e per notifiche altri problemi comuni
Analisi Dinamica		
Jest	27.5.1	Framework javascript per il testing, creato appositamente per React
Chai	4.3.6	Libreria javascript per il testing
Mocha	9.2.2	Libreria javascript per il testing

Tabella 6: Strumenti per l'analisi e l'integrazione del codice

6 Architettura

Per lo sviluppo generale dell'applicazione, è stata seguita una architettura DApp$_G$, (Fully) Decentralised Application. Essa consiste in un frontend Web che effettua chiamate dirette ad una infrastruttura decentralizzata backend, nel nostro caso una gerarchia di smartcontracts eseguiti su blockchain Fantom; questa struttura ricorda quindi una architettura client-server, senza un supporto intermedio per le operazioni.
Il pattern architetturale$_G$scelto dal gruppo per lo sviluppo del frontend è il Model-View-ViewModel$_G$. Il seguente pattern è tra i più diffusi nello sviluppo delle web application e permette di scrivere codice facilmente mantenibile e riusabile; questo è possibile grazie al forte disaccoppiamento che sussiste tra logica di presentazione e di business. Inoltre l'MVVM è risultato il più adatto per essere utilizzato con React, libreria impiegata per lo sviluppo dell'UI e che renderizza le componenti in base al loro stato$_G$ interno.

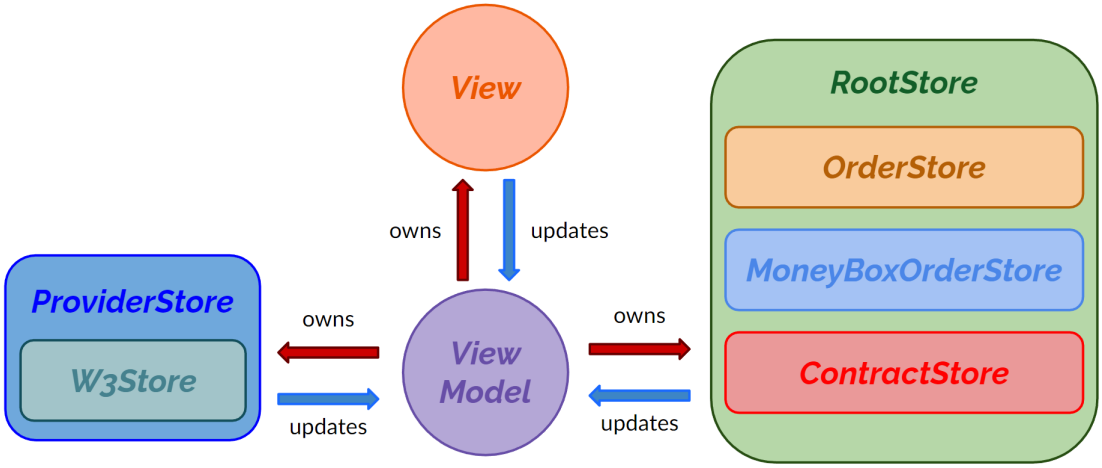

Figura 8: Model-View-ViewModel di ShopChain

Il passaggio dei dati dal Model alle varie componenti grafiche avviene attraverso l'utilizzo di un Context React, al quale viene passato un'istanza del ViewModel. L'utilizzo di un Context React ci permette di accedere al valore corrente del ViewModel in qualsiasi porzione della View, senza doverlo passare di componente in componente attraverso le props (ossia gli argomenti dei componenti che compongono la vista). Nella radice dell'applicazione viene infatti creata un'istanza del ViewModel, che viene passata ad un Context.Provider, che fa da contenitore per tutta la View. All'interno di tale contenitore ogni componente puo' utilizzare un hook per accedere al Context React ed utilizzare il valore più recente del ViewModel.
E' stato scelto di utilizzare un Context React per il passaggio dei dati in quanto la nostra applicazione è molto profonda e non risultava conveniente passare i dati per molti componenti rischiando, nel peggiore dei casi, di doverli utilizzare nell'ultimo della gerarchia. Per poter fare in modo che una componente della View si renderizzi non solo al cambiamento del suo stato interno ma anche al cambiamento dei dati nel Model, abbiamo utilizzato la libreria Mobx$_G$. Questa ci permette di implementare l'observer pattern$_G$, non supportato di default da React. A tale scopo, Mobx permette di segnare delle classi (o attributi di esse) come observable$_G$ e di costruire dei componenti della View come observer$_G$.
Quest'ultimi vengono automaticamente ri-renderizzati al cambiamento di un qualsiasi attributo observable.

6.1 Diagrammi delle classi

6.1.1 Classi Frontend

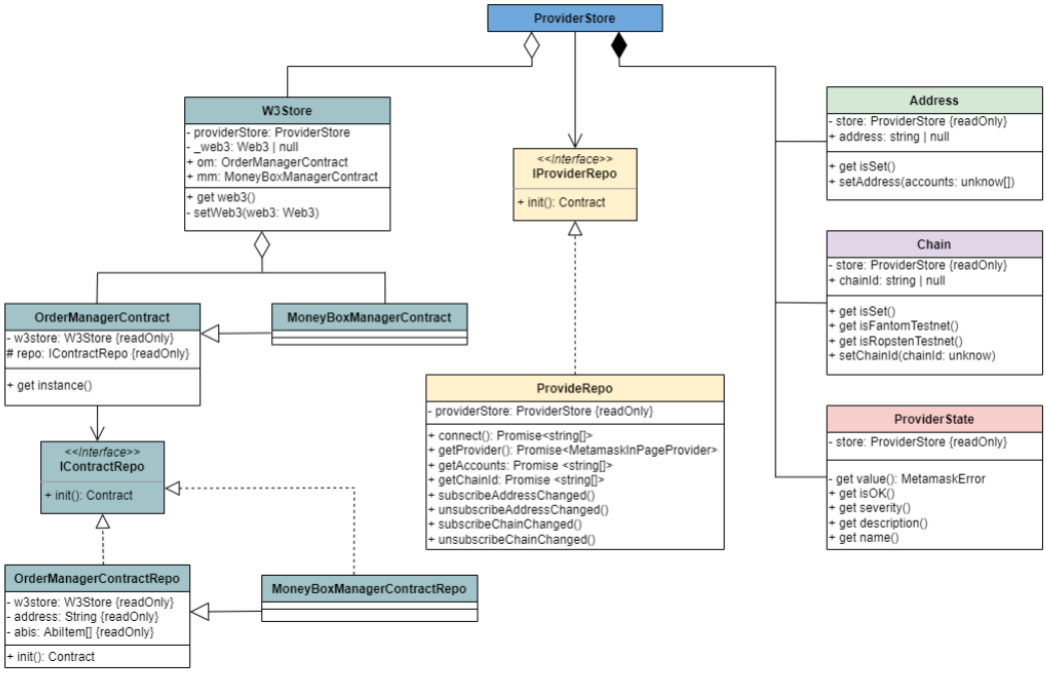

Figura 9: La gerarchia di ProviderStore

Si può notare la struttura ad albero degli store, in questo caso particolare il Provider Store, il quale da come si può vedere fa ampio uso di ereditarietà proprio come avviene a livello di smart contract. La classe singleton **ProviderStore** si occupa di gestire la connessione alla blockchain, tramite l'istanziazione di **W3Store**; l'istanziazione della connessione ai due contratti Order e Moneybox, l'istanziazione della connessione ai due contratti per ordini e moneybox, l'address del wallet collegato **Address**, la tipologia di blockchain a cui si è collegati **Chain** e, con tutte le precedenti, ne deriva inoltre uno stato generale **ProviderState**, con relativi errori.

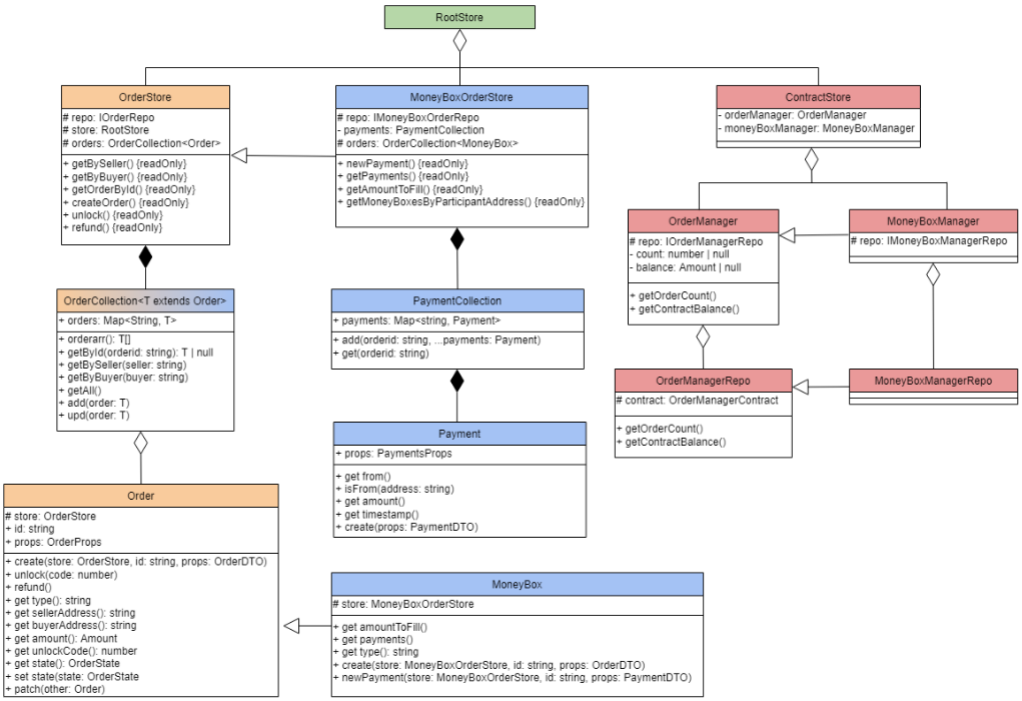

Figura 10: Rootstore, le classi per gli ordini e la gestione dei contratti

Anch'esso con struttura ad albero, RootStore si occupa di gestire i dati di dominio, gli ordini singoli e moneybox e i pagamenti. Ci sarà quindi **OrderStore** e classi relative che si occupa di gestire tutti gli ordini di base e **MoneyboxOrderStore** il quale estende OrderStore trattando non più ordini di base ma MoneyBox e aggiungendo la gestione dei pagamenti multipli. Stessa cosa accade a livello di **ContractStore**, ma in maniera più semplice in quanto la differenza effettiva sta nella repository assegnata: in base al caso base (order) o al caso derivato (moneybox) il resto delle chiamate rimarranno invariate.

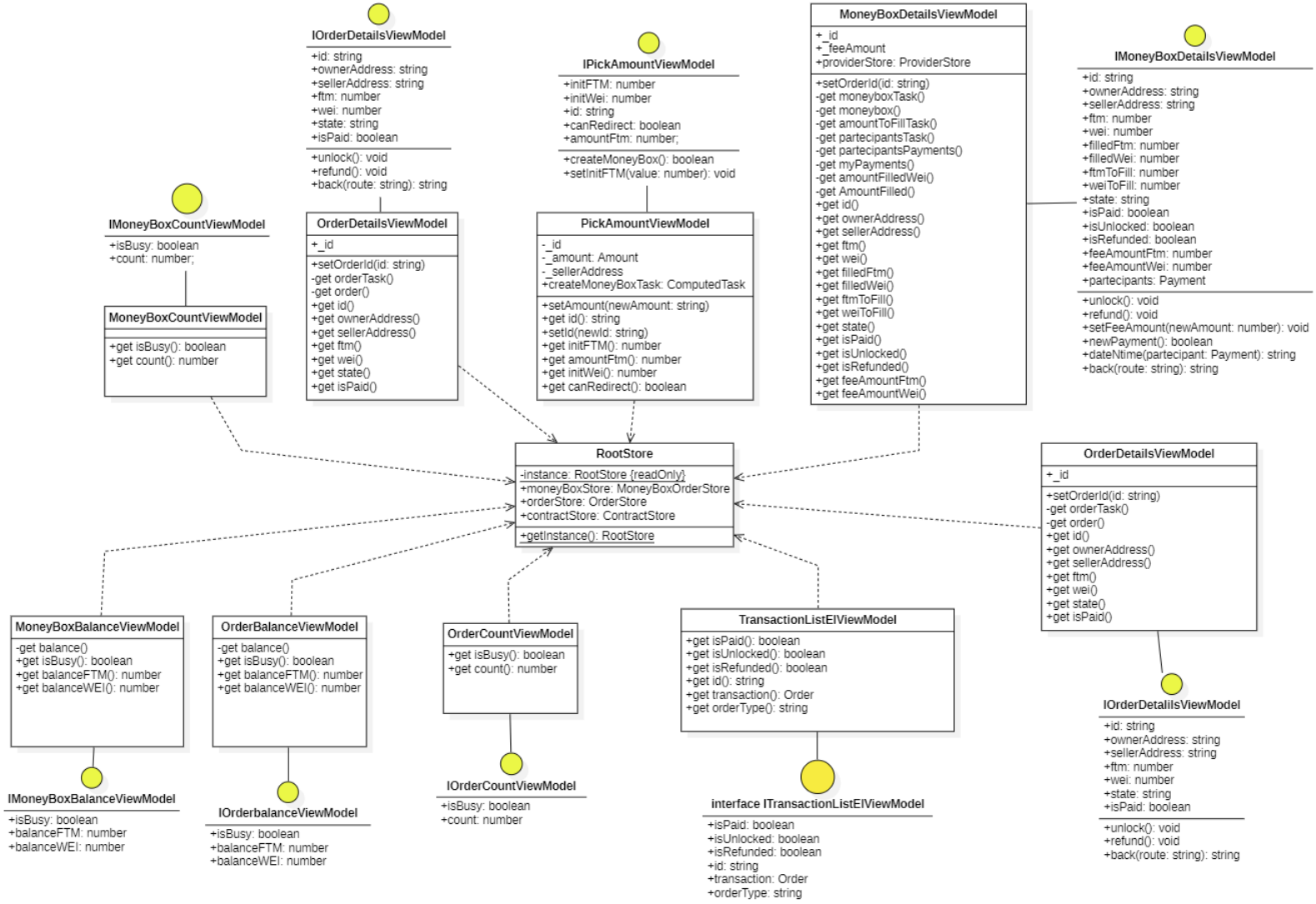

Figura 11: Le classi viewmodel dipendenti da RootStore

Da **RootStore**, dipendono poi la grande maggioranza delle classi ViewModel, ognuna con una sua interfaccia, che si occupano di gestire le chiamate dalla vista e apportare cambiamenti alla stessa.

6.1.2 Classi Solidity

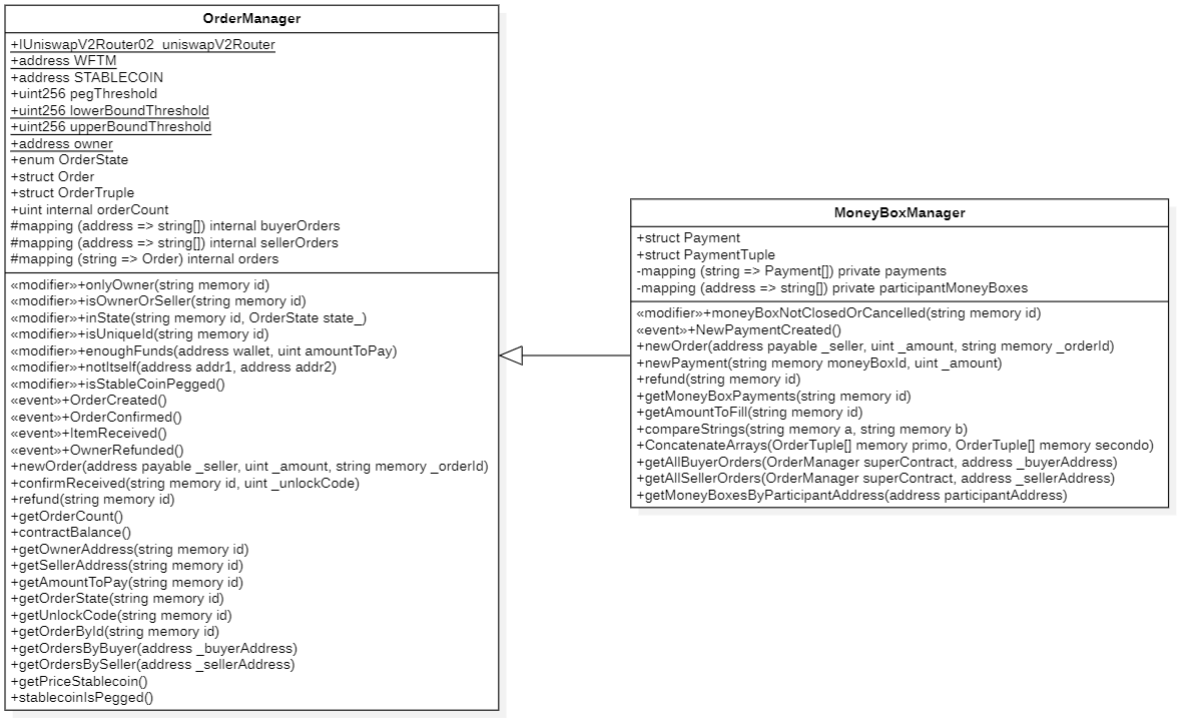

Figura 12: Le classi Solidity

Il diagramma delle classi Solidity, che gestiscono gli smartcontracts, è stato creato seguendo le indicazioni di **sol2uml**. Esso è costituito da due classi:

- **OrderManager**, si occupa della gestione di tutti i pagamenti singoli, mappati per una migliore gestione degli stessi;
- **MoneyBoxManager**, si occupa della gestione delle Moneybox, anche esse mappate; essendo un'estensione di OrderManager, ne eredita tutti i metodi.

I primi sei attributi di **OrderManager**, e le funzioni *isStableCoinPegged()*, *getPriceStableCoin()* e *stablecoinIsPegged()* servono alla conversione in stable coin_{G} del pagamento effettuato in FTM, per garantire una maggiore stabilità della transazione. Per maggiori informazioni sulla implementazione della conversione in questo progetto, si visiti la sezione dedicata §**??**.
Tra i metodi, *refund(string memory id)* si occupa di ritornare ai vari proprietari degli ordini (e a tutti i partecipanti nel caso di una MoneyBox) le somme di FTM interne al particolare contratto, identificato tramite *id*.
Per maggiori informazioni sulla parte generale dell'applicazione relativa alla Blockchain, si invita a consultare la sezione §7.

6.2 Diagrammi di sequenza

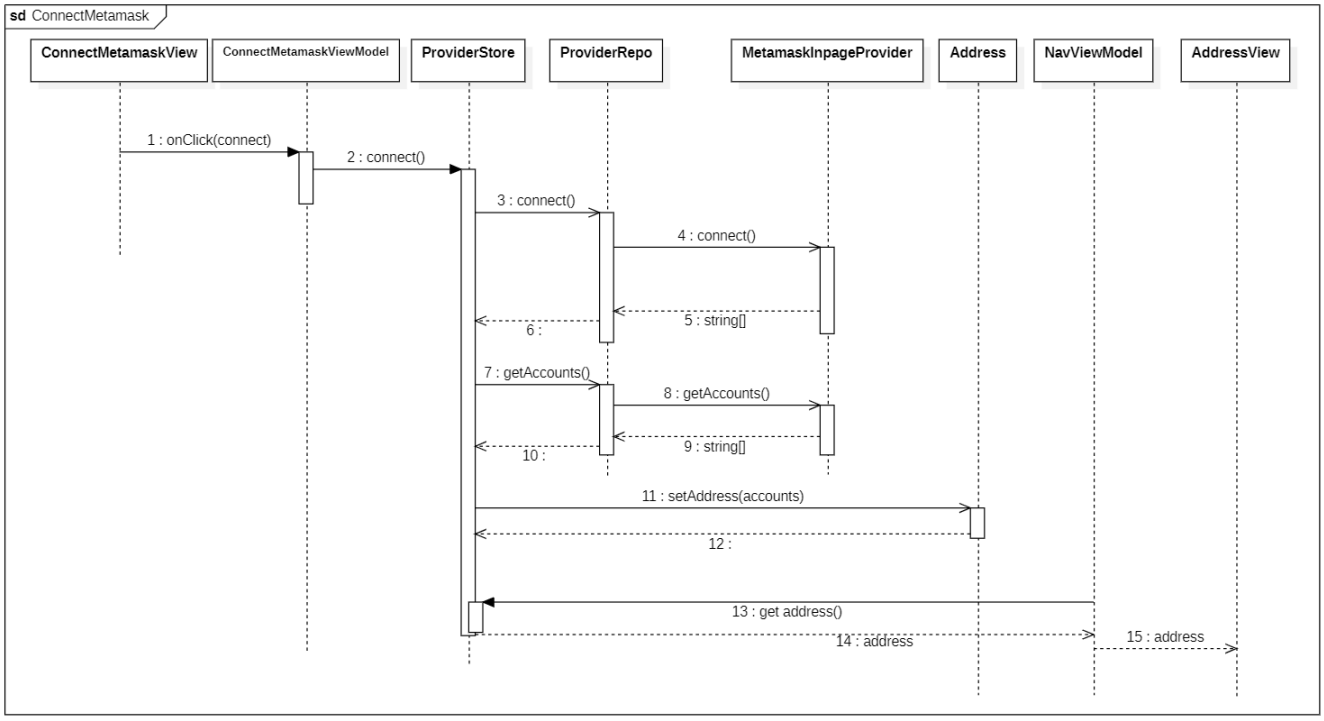

Figura 13: Diagramma di sequenza per la connessione a Metamask

Il diagramma di sequenza sopra riportato è cosi descritto:

1. La connessione a MetaMask inizia quando nella **ConnnectMetamaskView** mediante l'apposito pulsante viene fatto l'onclick e viene chiamato il metodo *connect()* sul **ConnectMetamaskViewModel**;

2. Esso a sua volta chiama il metodo *connect()* sul **ProviderStore** che a sua volta fa la chiamata asincrona del metodo *connect* sul **ProviderRepo** che a sua volta fa la chiamata asincrona del metodo *connect* su **MetamaskInpageProvider** e la connessione è così stabilita;

3. Successivamente dato che potrebbero esserci più account connessi il **ProviderStore** chiama il metodo asincrono *getAccounts* sul **ProviderRepo** che lo chiama a sua volta sul **MetaMaskInpageProvider**, il quale ritorna tutti gli indirizzi degli account connessi mediante un array di stringhe;

4. Succcessivamente il **ProviderStore** setta l'indirizzo mediante il metodo *setAddress* al quale viene passato in input il primo carattere dell'array di stringhe ritornato nel punto precedente;

5. Infine la **NavViewModel** prende gli indirizzi connessi mediante il metodo *getAddress* chiamato sul **ProviderStore**, e li ritorna alla **AddressView** per essere visualizzati.

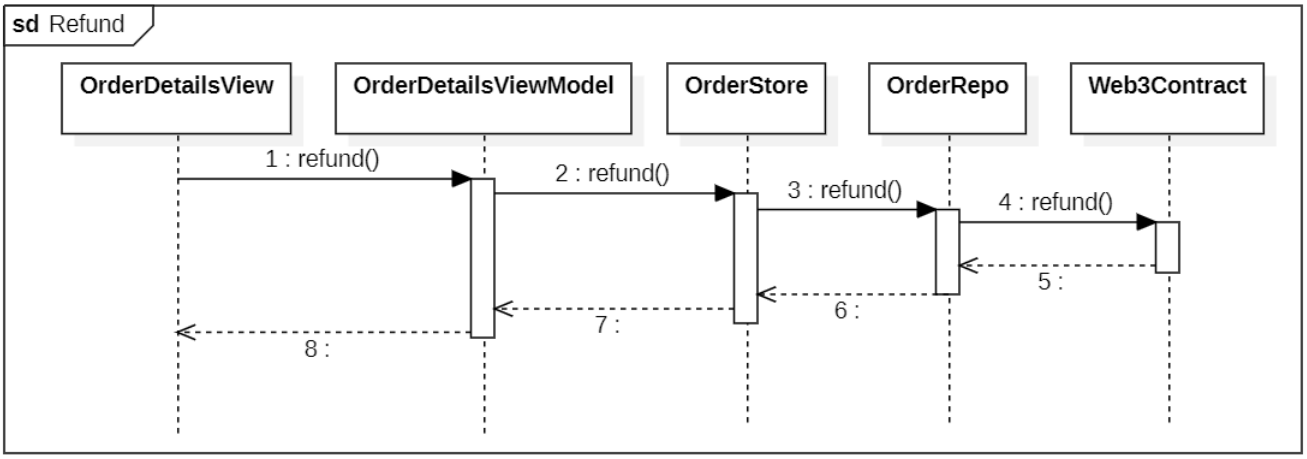

Figura 14: Diagramma di sequenza della funzione di rimborso

Il diagramma di sequenza sopra riportato, molto semplice grazie ai pattern architetturali utilizzati, è cosi descritto:

1. Dalla **OrderDetailsView**, dopo aver cliccato l'apposito bottone, parte la chiamata della funzione *refund()* verso **OrderDetailsViewModel** per effettuare il rimborso dell'ordine corrente;
2. Dal view-model, vengono chiamate le funzioni *refund()* fino ad arrivare a Web3Contract, che si occuperà di contattare lo smartcontract nella Blockchain;
3. Al ritorno delle varie funzioni, l'utente visualizzerà un messaggio di avvenuto rimborso dei pagamenti.

6.3 Architettura di dettaglio

6.3.1 Repository pattern

Il repository pattern è una astrazione del Data Access Layer (DAL_G). Nasconde i dettagli su come esattamente i dati vengono salvati o recuperati dall'origine dati sottostante.
I dettagli sul modo in cui i dati vengono salvati e recuperati si trovano nel rispettivo repository. Il repository pattern ci ha permesso di racchiudere tutte le interazioni il Data Access Layer (nel nostro caso la blockchain) in un oggetto unico, mantenendo la separazione tra implementazione e interfaccia. Tali Repository verranno utilizzati all'interno degli store, tramite Dependency $Injection_G$.

6.3.2 Singleton

Sono state implementate alcune classi seguendo il design pattern Singleton, ciò consente di garantire che una classe abbia una sola istanza, fornendo al contempo un punto di accesso globale alla stessa. Il design singleton è stato usato per avere un'unica fonte detentrice dello stato di ogni modello. Le classi più importanti create seguendo il pattern singleton sono:

- **RootStore**: classe che gestisce i dati di dominio, gli ordini, le moneybox e i pagamenti;
- **ProviderStore**: classe che gestisce la connessione a blockchain e tutte le classi relative, e l'istanziazione di W3Store;
- **W3Store**: classe che gestisce la connessione a Metamask.

6.3.3 Observer pattern

Attraverso Mobx abbiamo usato l'observer pattern in quasi tutte le classi del frontend in modo tale che, alla modifica di una classe observable, i cambiamenti si rifletteranno nelle classi observer.

7 Blockchain

La nostra applicazione non fa uso di alcun backend in quanto sfrutta appieno le funzionalità della blockchain: i contratti che svolgono la funzione di business logic dell'applicazione sono stati caricati sulla testnet$_G$ Fantom. Tutti i file dedicati alla blockchain si trovano nel repository **smart-contract**, consultabili e clonabili al seguente link:

https://github.com/Yakuzaishi-SWE/smart-contract

I file sono cosi organizzati:

- *test* è la sottocartella contenente i file **1_OrderManager.test.js** e **2_MoneyBox.test.js**, utili a verificare che i contratti siano corretti;
- *contracts* è la sottocartella contenente i due smartcontract **OrderManager.sol** e **MoneyBoxManager.sol**;
- **hardhat.config.js**, locato nella cartella principale, è il file che configura la connessione alla testnet.

8 Conversione a Stable Coin

La funzionalità di convertire l'ammontare depositato sul contratto in stable coin è un'interessante funzionalità che il gruppo ha individuato ed aggiunto ai requisiti.
Il gruppo ha condotto quindi uno studio relativamente ai passaggi da affrontare per attuare il meccanismo di conversione dei token$_G$. Prima di procedere è necessario descrivere e comprendere alcune componenti fondamentali che sono necessarie:

- Wrapped FTM: è un token crypto ancorato al valore del Fantom. Viene chiamato così oerchè l'asset originale viene messo in un wrapper che consente di crearne una versione su un'altra blockchain. Questo permette di creare ponti tra diverse blockchain e utilizzare asset non nativi su un'altra blockchain. Ovviamente in genere può essere riscattato per il corrispettivo in qualsiasi momento. Il processo di "wrapping" permette quindi di rendere il token Fantom conforme allo standard ERC-20 e poter essere, ad esempio nel nostro caso, convertito;
- Stable Coin: è un token che ha valore fissato con un rapporto 1:1 con una valuta FIAT$_G$;
- Liquidity Pool: è un insieme di fondi bloccati in uno smart contract che generalmente contengono due token diversi. Hanno diversi casi d'uso tra cui la possibilità di fungere da punto di scambio tra i due token;
- Factory Contract: è un contratto che si occupa di gestire ed eseguire oprazioni sulle coppie di token. In particolare la sua funzione principale è quella di creare uno smart contract associato alla coppia di token scelta. Quest'ultima funzione è necessaria se si vuole generare una propria liquidity pool;
- Router Contract: è un contratto che fornisce un supporto a tutte quelle che sono le funzionalità di trading dei token e gestione della liquidità.

Le componenti citate sono fondamentali per poter effettuare la conversione in stable coin. Il gruppo ha deciso di sviluppare un primo PoC di questa funzionalità direttamente sulla rete fantom di test. La rete di test presenta però delle mancanze rispetto alla rete principale, vengono a mancare infatti i contratti di alcuni token, come ad esempio USDT (scelto dal gruppo), e quelli che si occupa di gestire le liquidity pool di WFTM e eventuali stable coin. Il gruppo ha deciso di creare un token che simuli il comportamento di USDT (è possibile visualizzare il contratto associato al seguente link):
link
Purtroppo non è possibile replicare il funzionamento di un vero USDT e il suo relativo vincolo al valore del dollaro, ma è sufficiente per dimostrare come avvenga il processo di conversione.

Sulla rete Fantom il provider dei contratti per la gestione delle coppie di token e delle pool di liquidità è SpookySwap. In particolare come anticipato, è necessario riferirsi ai due contratti UniswapV2Factory (Contract Factory) e UniswapV2Router02 (Contract Router), è possibile visualizzarne i riferimenti al seguente link della documentazione:

https://docs.spooky.fi/Resources/contracts

Tramite il contratto UniswapV2Factory abbiamo creato la pair WFTM/USDT (usando il token ERC-20 da noi creato).
Dopo aver creato la coppia è stata generata la corrispondente liquidity pool in cui è possibile fornire o rimuovere liquidità. è importante ricordare che il valore del token USDT da noi creato è associato al rapporto delle quantità dei due token presenti nella pool e non al dollaro.

A questo punto abbiamo proceduto ad inserire all'interno del contratto la chiamata al contratto UniswapV2Router02 per effettuare la conversione tra i due token. Il metodo da chiamare è UniswapV2Router02.swapETHForExactTokens, che prende in input:

- Equivalente di USDT da richiedere alla pool;
- Array contente gli indirizzi dei due contratti dei due token;
- Indirizzo a cui ritornare gli USDT richiesti (in questo caso è necessario indicare l'indirizzo del contratto);
- Un timestamp che indichi il timeout dopo il quale la transazione deve essere annullata.

Per approfondire ulteriormente il metodo, si veda la relativa documentazione:

https://docs.uniswap.org/protocol/V2/reference/smart-contracts/router-02#swaptokensforexacttokens

Poichè la conversione viene effettuata dal router su una pool, che verrà utilizzata da più utenti, il valore della conversione può cambiare rapidamente anche solo di poco. Il router in tal caso tornerà un ammontare di FTM da restituire al mittente della transazione nel caso in cui la conversione sia a suo sfavore. è necessario tenere traccia di questo comportamento e gestire la restituzione di tale importo.

9 Punti di estensione

ShopChain è un'applicazione progettata pensando anche ad eventuali operazioni future di manutenzione o estensione del prodotto: durante lo sviluppo dell'applicazione ShopChain sono stati esplorati vari spunti per nuove features e ampliamenti, che però a causa del poco tempo rimasto non sono state implementate in tempo. Sono state individuate principalmente tre aree che potranno essere ampliate.

9.1 TheGraph protocol

Nella pagina di visualizzazione degli ordini il gruppo ha implementato la possibilità di filtrare gli ordini restituiti direttamente dalla blockchain. Attualmente questa funzionalità è fornita solo tramite un controllo a front-end, in futuro si potrebbe evolvere la funzione di filtraggio tramite l'implementazione del protocollo TheGraph$_G$.

The Graph è un protocollo decentralizzato per indicizzare e successivamente effettuare query dei dati presenti in blockchain in una modalità simile a quella che avviene con una normale base di dati. Questo permette di interrogare la blockchain in modo facile e veloce, ed estraendo dati che altrimenti non sarebbe possibile visualizzare.

In particolare quanto citato è possibile tramite la creazione di subgraph. Un sottografo non è altro che una serie di parametri necessari al protocollo per poter mappare e creare un indice dei dati presenti in blockchain.
Si compongono di tre componenti principali:

- Manifest (subgraph.yaml) che definisce quali dati il sottografo andrà a indicizzare;

```yaml
specVersion: 0.0.2
schema:
  file: ./schema.graphql
dataSources:
  - kind: ethereum
    name: OrderManager
    network: ropsten
    source:
      address: "0x50fA082C40e8409A6D8a9cB69d7E2DC82F717A05"
      abi: OrderManager
      startBlock: 12000000
    mapping:
      kind: ethereum/events
      apiVersion: 0.0.5
      language: wasm/assemblyscript
      entities:
        - ItemReceived
        - OrderConfirmed
        - OrderCreated
        - OwnerRefunded
      abis:
        - name: OrderManager
          file: ./abis/OrderManager.json
      eventHandlers:
        - event: ItemReceived(string,address,address,indexed uint256,indexed uint256,indexed uint8)
          handler: handleItemReceived
        - event: OrderConfirmed(string,address,address,indexed uint256,indexed uint256,indexed uint8)
          handler: handleOrderConfirmed
        - event: OrderCreated(string,address,address,indexed uint256,indexed uint256,indexed uint8)
          handler: handleOrderCreated
        - event: OwnerRefunded(string,address,address,indexed uint256,indexed uint256,indexed uint8)
          handler: handleOwnerRefunded
      file: ./src/mapping.ts
```

Figura 15: Esempio di un possibile subgraph configurato per il contratto OrderManager per la rete Ropsten.

- Schema (schema.graphql) che riporta quali dati si desiderà ricevere dal sottografo;

```
type Order @entity {
  id: ID!
  sellerAddress: Bytes!
  ownerAddress: Bytes!
  amount: BigInt!
  datetime: BigInt!
  state: Int!
}
```

Figura 16: Esempio di file schema per la definizione entità Order.

- AssemblyScript Mappings (mapping.ts) file che riporta la traduzione dei dati presenti in blockchain che il sottografo dovrà indicizzare.

```
export function handleOrderConfirmed(event: OrderConfirmed): void {

  let entity = Order.load(event.transaction.hash.toHex())

  if(entity == null) {
    entity = new Order(event.transaction.hash.toHex())
  }

  entity.sellerAddress = event.params.sellerAddress
  entity.ownerAddress = event.params.ownerAddress
  entity.amount = event.params.amount
  entity.datetime = event.params.datetime
  entity.state = event.params.state

  entity.save()
}

export function handleOrderCreated(event: OrderCreated): void {
  let entity = Order.load(event.transaction.hash.toHex())

  if(entity == null) {
    entity = new Order(event.transaction.hash.toHex())
  }

  entity.sellerAddress = event.params.sellerAddress
  entity.ownerAddress = event.params.ownerAddress
  entity.amount = event.params.amount
  entity.datetime = event.params.datetime
  entity.state = event.params.state

  entity.save()
}

export function handleOwnerRefunded(event: OwnerRefunded): void {
  let entity = Order.load(event.transaction.hash.toHex())

  if(entity == null) {
    entity = new Order(event.transaction.hash.toHex())
  }

  entity.sellerAddress = event.params.sellerAddress
  entity.ownerAddress = event.params.ownerAddress
  entity.amount = event.params.amount
  entity.datetime = event.params.datetime
  entity.state = event.params.state

  entity.save()
}
```

Figura 17: Esempio di file mapping per l'indicizzazione degli eventi.

Questa funzionalità non è stata portata avanti perchè non è disponibile sulla testnet di Fantom, ma solo sulla mainnet. Se la si vuole implementare sarà quindi necessario eseguire lo sviluppo e i relativi test su una testnet di Ethereum in cui è disponibile il protocollo (es. $\text{Rinkeby}_{\text{G}}$). Una volta che lo sviluppo sarà finito si potrà procedere a caricare il subgraph creato sulla mainnet di Fantom senza particolari modifiche, poichè quest'ultima è una blockchain EVM.

9.2 Chain differenti

Un ovvio ampliamento per l'applicazione è il supporto a più blockchain, per raggiungere un bacino di utenza più ampio e garantire quindi un maggior successo dell'applicazione.

9.3 Frontend

9.3.1 Immagine MoneyBox

Il gruppo aveva pensato di associare ad ogni MoneyBox una immagine generata automaticamente, o presa da un pool di immagini definito. Se si sceglie di generarla automaticamente, si può prendere come seed di generazione uno tra i seguenti: l'id dell'ordine associato alla MoneyBox, il numero del blocco relativo alla transazione in blockchain, il timestamp della creazione. Questo contribuisce ad una maggiore sicurezza nel condividere il link alla suddetta MoneyBox con amici, che potranno vedere a colpo d'occhio se il link è corretto tramite l'immagine incorporata.

9.3.2 Riflettere i cambiamenti del contratto

Con i cambiamenti che possono venire apportati agli smartcontract è bene che il lato frontend dell'applicazione sia aggiornato di conseguenza. Nel caso dei cambiamenti proposti finora i cambiamenti individuati sono i seguenti:

- **TheGraph**: le due pagine di elenco transazioni dovranno essere modificate per sfruttare appieno tutte le funzionalità portate dalla implementazione dei protocolli TheGraph;
- **Chain differenti**: in un pop-up all'ingresso dell'applicazione, l'utente potrà selezionare la chain (e quindi la cryptovaluta correlata) con la quale procedere al pagamento. Tale scelta dovrà essere riportata anche nella breadcrumb come reminder testuale.

Glossario

A

API

Le API, Application Programming Interface, sono connessioni tra diversi computers o tra diversi programmi, e definiscono come le parti comunicano tra loro tramite richieste e risposte.

B

BIOS

È il firmware utilizzato per fornire servizi runtime ai sistemi operativi e programmi, e per eseguire l'Inizializzazione hardware durante lo start-up della macchina.

Blockchain

Letteralmente significa "catena di blocchi", è una struttura dati decentralizzata i cui dati sono raggruppati in blocchi concatenati in ordine cronologico e la cui integrità è garantita dall'uso della crittografia. Il suo contenuto è "immutabile". Una volta scritto tramite un processo normato non è più modificabile né eliminabile, a meno di non invalidare l'intero processo.

C

Criptovaluta

Traduzione in italiano del termine inglese cryptocurrency e si riferisce ad una rappresentazione digitale di valore basata sulla crittografia. Si tratta di una risorsa digitale paritaria e decentralizzata. Al mondo esistono oltre 13000 criptovalute.

D

DAL

Un Data Access Layer (DAL) è un layer di un programma che fornisce un accesso semplificato ai dati memorizzati in una memoria persistente di qualche tipo, come ad esempio un database entità-relazionale.

DApp

È un'applicazione che può operare autonomamente, tipicamente tramite l'uso di smartcontracts eseguiti su una blockchain completamente decentralizzata.

Dependency Injection

In ingegneria software, la dependency injection è un design pattern in cui un oggetto contiene altri oggetti da cui dipende.

Docker

È una piattaforma open source di "containerizazzione": permette agli sviluppatori di racchiudere applicazioni in "container", componenti eseguibili standardizzati che uniscono il codice sorgente dell'app con le librerie di sistema e le dipendenze richieste per eseguire quel codice in qualsiasi ambiente.

E

E-Commerce

Insieme di attività di vendita e acquisto di prodotti effettuato tramite internet.

F

Fantom

Blockchain relativamente recente nata nel dicembre 2019 e si basa su un proprio algoritmo di consenso personalizzato. Vanta come punto di forza la velocità relativa alle transazioni, dichiara inferiore ai due secondi dagli sviluppatori stessi.

FIAT

Le FIAT sono valute emesse da un governo che sono supportate da un corrispettivo ammontare di un bene tangibile, molto spesso l'oro.

FTM

Simbolo di valuta di Fantom.

G

Git

Software di controllo di versione distribuito utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005.

GitHub

Servizio di hosting per sviluppatori. Fornisce uno strumento di controllo versione e permette lo sviluppo distribuito del software.

GraphQL

Linguaggio open source per richieste e manipolazione di dati per $\text{APIs}_{\mathbf{G}}$, è stata rilasciata pubblicamente nel 2015.

M

Metamask

Metamask è un software wallet per criptovalute usato per interagire con la blockchain Ethereum. Esso lascia accedere gli utenti ai loro walllet Ethereum tramite un'estensione browser o una app mobile, che sono poi usate per interagire con applicazioni decentralizzate.

MobX

È una libreria per gestire in modo reattivo gli stati di una applicazione. Nonostante sia usabile in modo indipendente da React, sono comunemente usati assieme.

Model-View-Controller(MVC)

Pattern architetturale molto diffuso nello sviluppo di software. Consiste nel dividere il prodotto in tre parti fondamentali: il modello, il controller, e la vista. Il controller si occupa di far comunicare vista e modello, ricevendo gli input dell'utente attraverso la vista e avvisando il modello dei cambiamenti avvenuti. La view in questo caso è in grado di visualizzare direttamente i dati contenuti nel modello.

Model-View-ViewModel(MVVM)

Pattern architetturale derivato dal Model-View-Controller(MVC)$_{\mathbf{G}}$ che prevede l'utilizzo di un View-Model per far comunicare vista e modello. In questo caso la vista non ha alcuna comunicazione con il modello quindi il disaccoppiamento è maggiore rispetto a MVC.

O

Observable

È l'oggetto osservato nel design pattern Observer$_{\mathbf{G}}$, detto anche "Subject".

Observer

È l'oggetto osservatore nel design pattern Observer$_{\mathbf{G}}$.

Observer design pattern

È un design pattern software utilizzato per far si che un oggetto observable$_{\mathbf{G}}$ ("osservato") notifichi modifiche interne all'observer$_{\mathbf{G}}$ ("osservatore"), che reagirà di conseguenza, solitamente chiamando un suo metodo interno.

P

Pattern architetturale

È una modellazione architetturale di un prodotto software. Ne esistono diversi e permettono di separare il comportamento delle componenti che lo compongono, aumentando il disaccoppiamento e favorendo lo unit-test delle varie componenti.

R

Repository

Ambiente di un sistema informativo, in cui vengono gestiti i metadati, attraverso tabelle relazionali; l'insieme di tabelle, regole e motori di calcolo tramite cui si gestiscono i metadati prende il nome di metabase.

Rinkeby

Una delle due maggiori testnet$_{\mathbf{G}}$ Ethereum.

S

Smart contract

Sono essenzialmente dei programmi salvati sulla blockchain che convertono i tradizionali contratti nella loro controparte digitale. Hanno quindi il vero e proprio obiettivo di digitalizzare i termini di un accordo in codice che viene eseguito quando i termini del contratto vengono rispettati.

Stable Coin

Una criptovaluta con un valore che riflette completamente quello di una valuta reale, molto spesso il dollaro, per garantire una maggiore stabilità nelle transazioni.

Stato interno

Un componente React può possedere uno stato interno (statefull) oppure no (stateless). Questo stato è un insieme più o meno numeroso di dati, i quali possono essere modificati attraverso interazioni dell'utente con questo componente (nella vista). La modifica di tali dati causa la rirenderizzazione del componente stesso e dei suoi figli.

T

Testnet

Nel gergo crypto, una testnet è una istanza di una blockchain usata per testing e sperimentazioni, senza il rischio di pagamenti e di intaccare la rete principale. Le cryptovalute delle testnet sono diverse da quelle delle reti principali, non hanno valore, e sono gratuitamente ottenibili.

TheGraph

È un protocollo di indicizzazione per organizzare dati blockchain e renderli facilmente accessibili tramite $\text{GraphQL}_{\text{G}}$.

Token

È uno standard per token fungibili, la cui proprietà è che ogni token è esattamente uguale per tipo e valore a un altro token.

Token

È un asset digitale, ad esempio una cryptovaluta.

W

Wallet

Letteralmente un portafoglio per le criptovalute, che si traduce fisicamente in un dispositivo, un supporto fisico, un programma o un servizio che memorizza le chiavi pubbliche e/o private per le transazioni di criptovaluta.