



OBERON

SPECIFICA ARCHITETTURALE V. 1.0.0

A.A. 2021-2022

Componenti del gruppo:

Casazza Domenico, matr. 1201136

Casonato Matteo, matr. 1227270

Chen Xida, matr. 1217780

Pavin Nicola, matr. 1193215

Poloni Alessandro, matr. 1224444

Scudeler Letizia, matr. 1193546

Stojkovic Danilo, matr. 1222399

Indirizzo repository GitHub:

<https://github.com/TeamOberon07/ShopChain>



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Registro delle modifiche

v	Data	Nominativo	Ruolo	Descrizione
1.0.0	17/05/2022	Casazza Domenico	Responsabile	Approvazione del documento
0.4.0	16/05/2022	Poloni Alessandro	Verificatore	Verifica del documento
0.3.6	15/05/2022	Stojkovic Danilo, Chen Xida	Progettista, Programmatore	Diagrammi di attività
0.3.5	14/05/2022	Casonato Matteo	Programmatore/Progettista	Stesura Descrizione SmartContract §(3.5), Modifica Possibili punti di estensione §(4), Stesura funzioni SmartContract §(5.1)
0.3.4	14/05/2022	Casazza Domenico	Progettista	Descrizione metodi Landing Page §(5.3), correzione diagrammi di sequenza §(3.2.2) e §(3.3.2)
0.3.3	13/05/2022	Casazza Domenico	Progettista	Descrizione metodi webApp §(5.2)
0.3.2	12/05/2022	Casazza Domenico	Progettista	Descrizione diagramma delle classi webApp §(3.2.1) e Landing Page §(3.3.1)
0.3.1	11/05/2022	Stojkovic Danilo	Progettista	Scrittura funzioni App Mobile
0.3.0	09/05/2022	Casazza Domenico	Amministratore	Verifica del documento e correzione errori di battitura
0.2.3	09/05/2022	Chen Xida	Progettista	Stesura diagrammi di sequenza §(3.2.2) e §(3.2.3)
0.2.2	05/05/2022	Chen Xida	Progettista	Stesura sezione funzioni §(4)
0.2.1	03/05/2022	Chen Xida	Progettista	Stesura punti di estensione §(5)
0.2.0	01/05/2022	Casonato Matteo	Verificatore	Verifica del documento
0.1.1	08/04/2022	Casazza Domenico	Amministratore	Ampliamento Pattern Architetturale §(3.1)

v	Data	Nominativo	Ruolo	Descrizione
0.1.0	05/04/2022	Casazza Domenico	Verificatore	Verifica del documento
0.0.1	05/04/2022	Chen Xida	Progettista	Creazione bozza documento §(1), §(2), §(3.1)

Indice

1	Introduzione	1
1.1	Scopo del documento	1
1.2	Obiettivi del prodotto	1
1.3	Riferimenti	1
1.3.1	Riferimenti informativi	1
1.3.2	Riferimenti tecnici	1
2	Architettura	2
2.1	Pattern architetturale	2
2.2	Design patterns	4
2.3	WebApp	5
2.3.1	Diagramma delle classi	5
2.3.2	Diagrammi di sequenza	7
2.3.2.1	Visualizzazione degli ordini	7
2.3.2.2	Richiesta di rimborso dell'ordine	8
2.3.3	Diagramma di Attività: Accesso HomePage	9
2.4	Landing Page	10
2.4.1	Diagramma delle classi	10
2.4.2	Diagrammi di sequenza	11
2.4.2.1	Creazione dell'ordine	11
2.5	App Mobile	12
2.5.1	Diagramma delle classi	12
2.5.2	Diagrammi di sequenza	14
2.5.2.1	Scannerizzazione QRCode	14
2.5.2.2	Visualizzazione ordini del buyer	16
2.6	Smart Contract	17
2.6.1	Diagramma delle classi	17
2.6.2	Diagramma di Attività: Ciclo di vita di un ordine	19
3	Possibili punti di estensione	20
3.1	Fees rimborsate dall'e-commerce	20
3.2	Ottimizzazione filtri	20
3.3	Sistema di reward	20
3.4	Verifica seller reali	21
3.5	Strategie di investimento per i fondi bloccati	21
4	Funzioni nella codifica	22
4.1	Smart Contract	22
4.2	WebApp	23
4.3	Landing Page	26
4.4	App Mobile	27

1 Introduzione

1.1 Scopo del documento

In questo documento si possono trovare i pattern architeturali sfruttati per lo sviluppo del prodotto. Nello specifico faremo riferimento a dei paper pubblicati negli ultimi anni, dato che dopo un periodo di ricerca il team ha constatato che non ci sono ancora dei veri e propri design pattern per le cosiddette DApp (Decentralized Application). Riportiamo inoltre i vari diagrammi che riassumono l'implementazione dei prodotti,

1.2 Obiettivi del prodotto

Al giorno d'oggi, numerosi sono gli e-commerce che non hanno un sistema affinché l'acquirente e il venditore possano creare transazioni sicure. Difatti, l'acquirente può venire truffato dal venditore se dopo il pagamento non gli viene consegnato il prodotto o viceversa.

ShopChain è un applicativo in grado di affiancare un e-commerce nelle fasi di pagamento fino alla consegna usando la tecnologia delle blockchain. La blockchain è incaricata di ricevere l'ammontare speso dall'acquirente in criptovaluta, consegnandola al venditore solo quando il pacco gli viene recapitato.

Nel momento della consegna del pacco l'acquirente dovrà necessariamente inquadrare il QR code applicato sul collo che ne certifica l'avvenuta consegna. Quindi verrà effettuato il passaggio della criptovaluta dal wallet della piattaforma al wallet del venditore.

1.3 Riferimenti

1.3.1 Riferimenti informativi

- È stato creato il documento *Glossario_1.0.0.pdf* per chiarire il significato dei termini tecnici che possono creare dubbi e perplessità.
- La pianificazione è divisa in sprint, seguendo la metodologia agile. Le modalità e il modello di sviluppo sono riportate nel documento *NormeDiProgetto_2.0.0.pdf*

1.3.2 Riferimenti tecnici

- Pattern architeturali 1: <https://medium.com/hexamount/architecting-modern-decentralized-applications-52b3ac3baa5a>
- Pattern architeturali 2: <https://ieeexplore.ieee.org/abstract/document/8432174>

2 Architettura

2.1 Pattern architetturale

Dopo un approfondito periodo di ricerca il team ha individuato l'architettura più opportuna tra quelle dedicate alle DApp basate su blockchain (sistema distribuito).

L'architettura di ShopChain è di tipo Fully Decentralized (Pure DApp).

In una Pure DApp l'utente, dopo essersi connesso al proprio wallet, dal front-end può chiamare direttamente i metodi dello SmartContract senza dover passare per un intermediario (con tutti i vantaggi e gli svantaggi che ne conseguono). Questo è possibile se il frontend viene hostato su servizio distribuito come ad esempio IPFS (InterPlanetary File System).

Vantaggi principali offerti dal pattern architetturale:

- Maggiore decentralizzazione (assenza di un server centralizzato)
- Maggiore sicurezza per l'utente (non ci sono intermediari tra di esso e la blockchain)

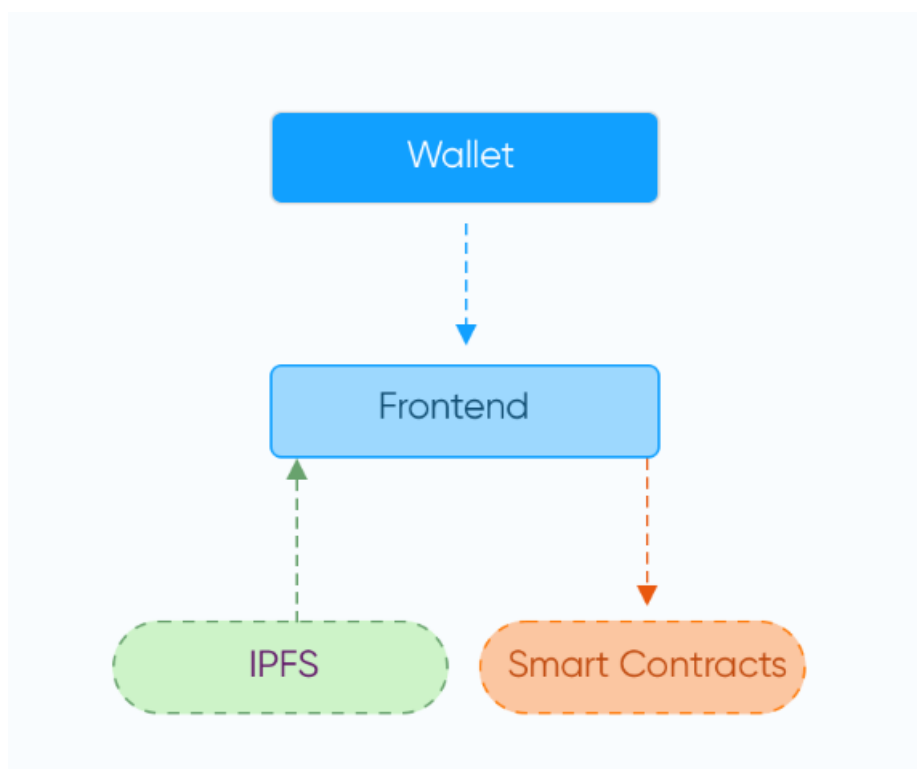


Figura 1: Interazione tra Wallet - Frontend - IPFS - Smart Contract

Fonti:

- <https://medium.com/hexmount/architecting-modern-decentralized-applications-52b3ac3baa5a>
- <https://ipfs.io/>

L'architettura di ShopChain fa riferimento al "Pattern B – Self-Confirmed Transactions" descritto nel paper "Engineering Software Architectures of Blockchain-Oriented Applications" di F. Wessling e V. Gruhn dell'Università di Duisburg-Essen.

In questo paper il Pattern B consiste nell'interazione dell'utente solo con un'applicazione web e/o un gestore di wallet (in questo caso MetaMask) per creare transazioni su una blockchain: le transazioni non vengono create direttamente dall'utente ma vengono generate dall'applicazione web e poi mandate manualmente al nodo della blockchain a cui l'utente è collegato.

Questo pattern bilancia sicurezza e facilità nell'interazione con la webApp perché creare transazioni manualmente è un'operazione difficile e realizzabile solo da utenti esperti, ma questo implica che chi interagisce con l'applicativo si fidi degli sviluppatori dato che la generazione di una transazione non è completamente trasparente.

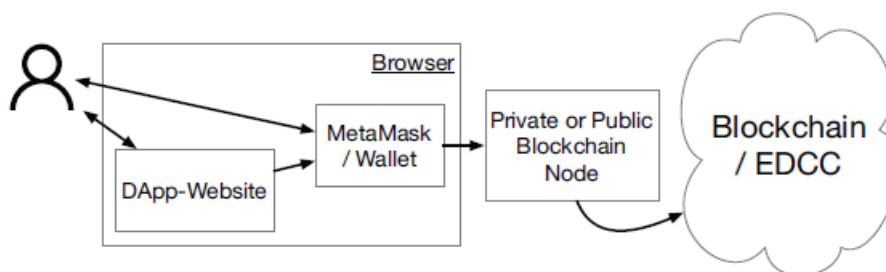
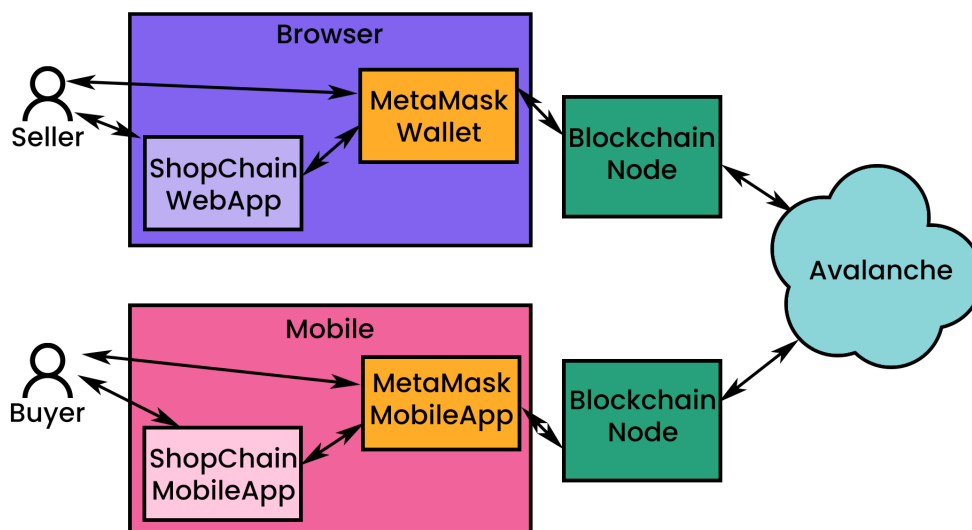


Figura 2: DApp Pattern B - Self-Confirmed Transactions

Fonte:

- <https://ieeexplore.ieee.org/abstract/document/8432174>

Applicando l'architettura appena descritta al progetto ShopChain si ottiene quindi:



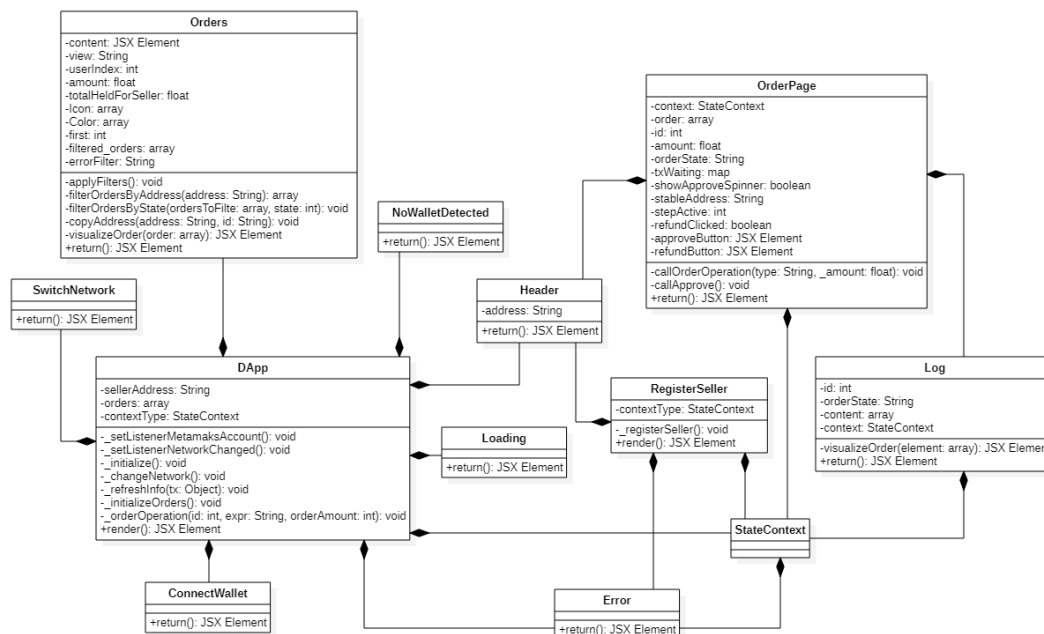
2.2 Design patterns

Nell'implementazione dell'applicativo sono stati utilizzati diversi design pattern per renderlo più sicuro e affidabile basandoci su schemi già testati:

- **Singleton**, applicato alla classe **stateContext** dell'app mobile per averne un'unica istanza accessibile da più classi;
- **React Hooks**, quali *useState*, *useEffect* e *useContext* per la visualizzazione dinamica delle informazioni della webApp;
- **Access Restriction**, per rendere accessibili determinate funzionalità solo ad una specifica tipologia di utente (ad esempio nella webApp per le corrispettive viste Buyer/Seller, e nello smart contract per garantire che alcuni parametri vengano modificati solamente dall'owner del contratto);
- **Guard Check**, per la verifica degli input inseriti dall'utente, sia dal punto di vista sintattico sia dal punto di vista semantico;
- **Presentational and Container Component**, per la realizzazione di componenti React che si occupano solo di renderizzare una vista (*Presentational*, e.g. SwitchNetwork, Loading, NoWalletDetected, ConnectWallet) e utilizzarli all'interno di un unico componente (*Container*, e.g. DApp e LandingPage);
- **Provider**, utilizzato per avere un componente accessibile globalmente all'interno della webApp;
- **Checks Effects Interactions**, nello smart contract, per evitare che altri smart contract malevoli abusino del control flow a seguito di una chiamata external (esempio: modifica dello stato di un ordine **prima** di effettuare il trasferimento del denaro);
- **Oracle**, per far sì che lo smart contract possa avere accesso ad informazioni esterne alla blockchain (nel nostro caso il prezzo della stablecoin), e possa fidarsi dei dati ricevuti (utilizziamo il servizio fornito da Chainlink);

2.3 WebApp

2.3.1 Diagramma delle classi



Il diagramma delle classi della webApp è costituito da tutti i componenti React che dialogano con la blockchain e compongono l'interfaccia utente. Partendo dall'alto della gerarchia il padre è **DApp**, il quale renderizza la pagina iniziale attraverso i componenti **Header** e **Orders**:

- **Header** si occupa di fornire all'utente le informazioni essenziali riguardanti il suo wallet, ovvero il balance e l'address, oltre al link per tornare alla pagina iniziale;
- **Orders** si occupa di mostrare all'utente una tabella contenente le informazioni degli ordini collegati al wallet connesso oltre alla possibilità di filtrare gli ordini per stato e per indirizzo.

All'interno della tabella creata da **Orders** è presente anche un bottone che rimanda al componente **OrderPage** che si occupa di far vedere all'utente i dettagli dell'ordine selezionato, il log di cambio stato e i bottoni per eseguire le operazioni sull'ordine:

- se l'utente è un compratore sarà presente il bottone di richiesta reso solo sugli ordini di cui non è già stato chiesto il reso;
- se l'utente è invece un venditore, sarà presente il QR code da applicare sul pacco al momento della spedizione, più i bottoni di cancellazione ordine, conferma richiesta reso e modifica dello stato dell'ordine in "Shipped" in base allo stato attuale dell'ordine.

I componenti **NoWalletDetected**, **SwitchNetwork** e **ConnectWallet** si occupano rispettivamente di avvisare l'utente che non ha installato l'estensione browser di MetaMask, di indirizzare l'utente sulla blockchain corretta nel caso in cui egli sia collegato ad una sbagliata e di collegare il wallet dell'utente alla webApp attraverso MetaMask.

StateContext

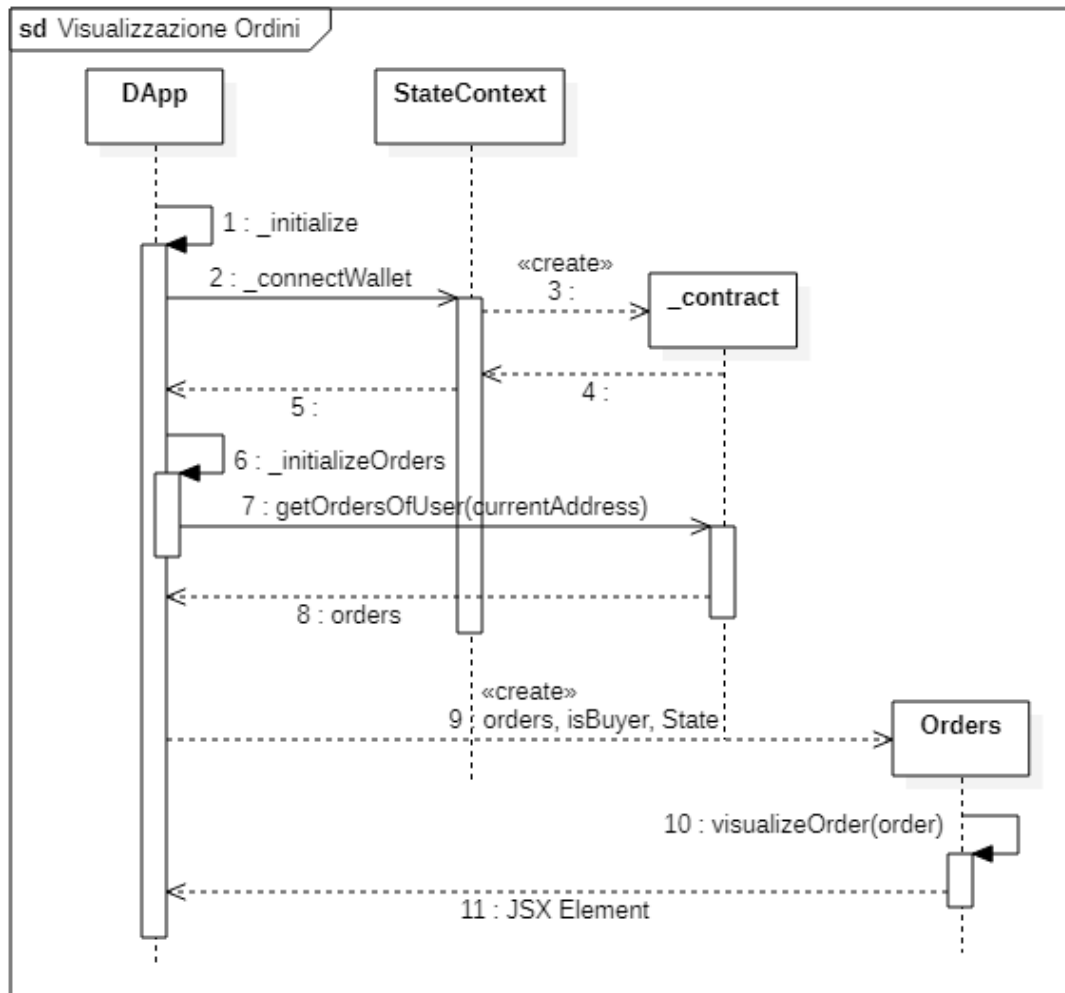
StateContext
+balance: float +contractAddress: String +stablecoinAddress: String +ourNetwork: String +rightChain: boolean +_contract: Contract +_provider: Web3Provider +userIsSeller: boolean +orderState: array +orderOperations: array +amountApproved: boolean +networks: array +orderStateChanged: boolean
+_connectWallet(): void +_changeNetwork(networkName: String): void +_wrongChain(): void +_rightChain(): void +_setListenerMetamaksAccount(): void +_setListenerNetworkChanged(): void +_updateBalance(): void +_getOrderById(id: int, order: Object): Object +_callCreateOrder(functionToCall: int, tokenAddress: String, orderAmount: float, maxAmountIn: float, sellerAddress: String, afterConfirm: function): String +_orderOperation(id: int, expr: String, orderAmount: float): array +_getSellers(): array +_userIsSeller(): void +_isAuthorizedSeller(sellerAddress: String): boolean +_getQRCode(): array +_getLog(id: int): array +_getERC20Balance(token: Object): float +_approveERC20(tokenAddress: String, amount: float): void +_ERC20isApproved(tokenAddress: String, amount: float): boolean +_setOrderStateChangedFalse(): void +_getAmountsIn(token: Object, amountOut: float): float +render(): JSX Element

StateContex è la classe React che si occupa di dialogare direttamente con lo smart contract e, quindi, con la blockchain: al suo interno infatti sono presenti i metodi per collegarsi allo smart contract, prendere da esso tutte le informazioni necessarie e gestire le operazioni sugli ordini. È l'unica classe a dialogare direttamente con lo smart contract.

Questa classe è vista come un componente React che contiene tutte quelle informazioni che devono essere condivise da vari altri componenti, per questo tutti i suoi attributi e metodi sono pubblici.

2.3.2 Diagrammi di sequenza

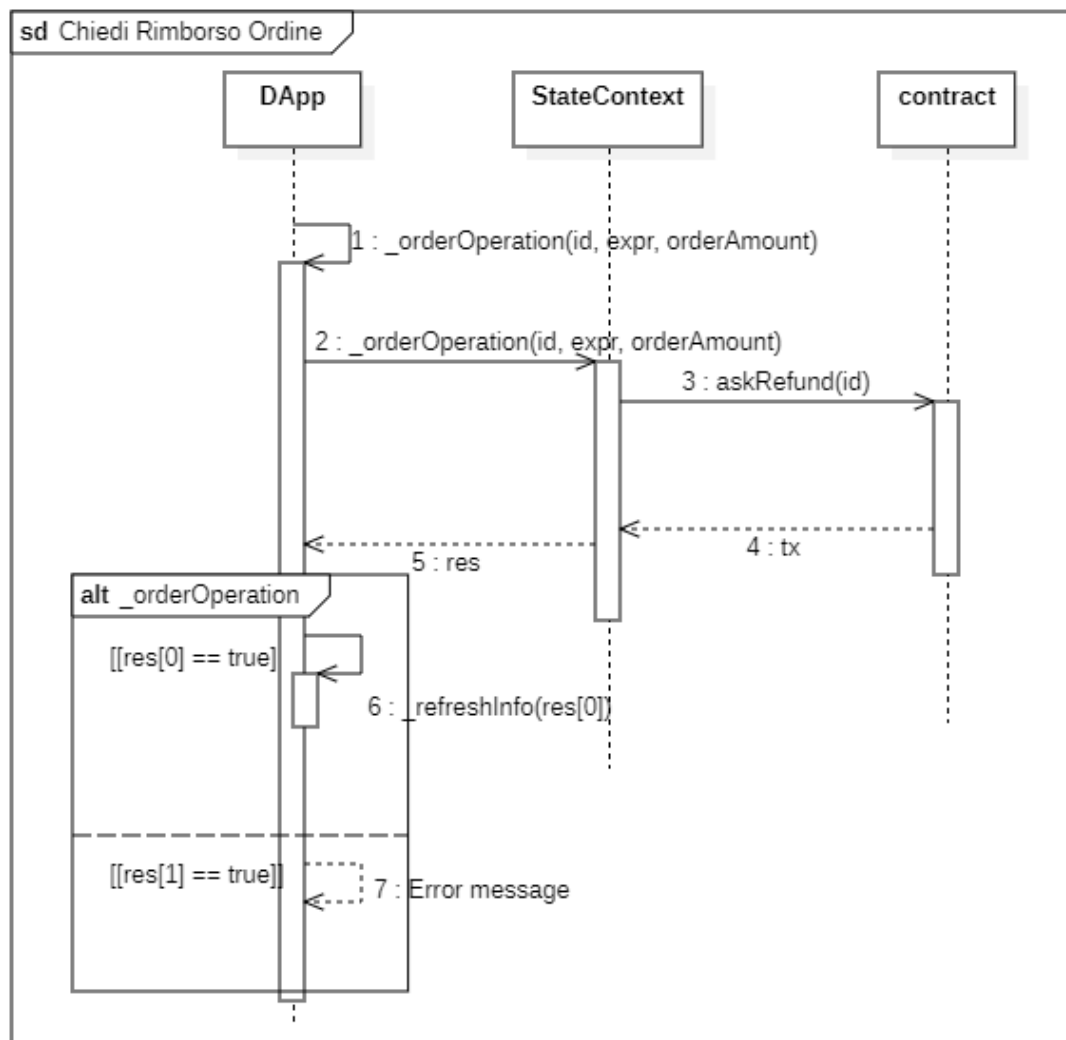
2.3.2.1 Visualizzazione degli ordini



Il diagramma descrive i passi per la visualizzazione degli ordini:

1. Viene invocata la funzione `_initialize()`, che a sua volta invoca la funzione `_connectWallet()` di `StateContext`;
2. Viene inizializzato lo smart contract e dopo ciò viene invocata la funzione `_initializeOrders()`;
3. Viene chiamata la funzione `getOrdersOfUser(currentAddress)` dello smart contract che ritorna la lista degli ordini dell'utente a `DApp`;
4. Infine viene creato il componente `Orders` che presa in input la lista degli ordini si occuperà di renderizzarli attraverso la funzione `visualizeOrder(order)`.

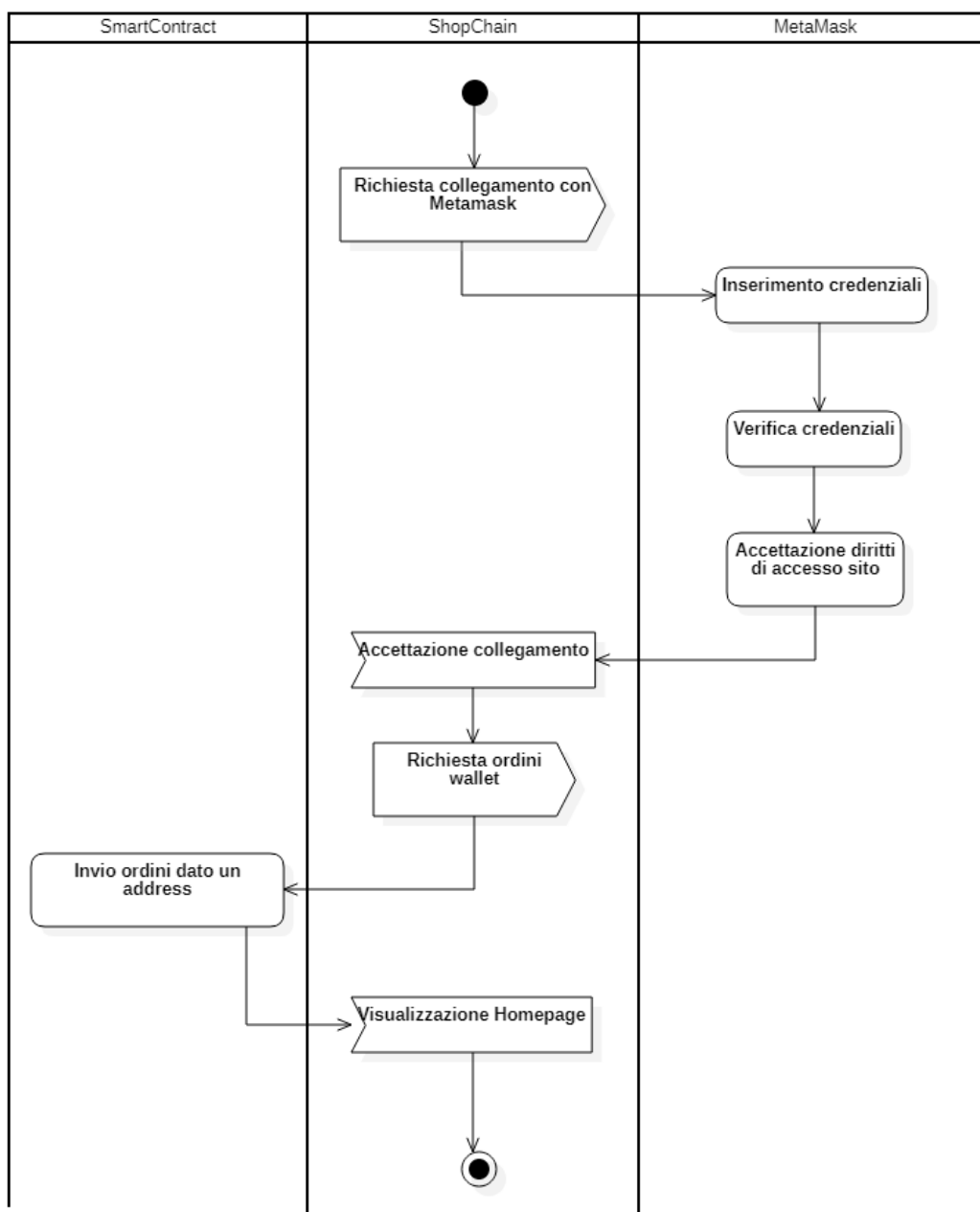
2.3.2.2 Richiesta di rimborso dell'ordine



Il diagramma descrive i passi per effettuare la richiesta di rimborso dell'ordine:

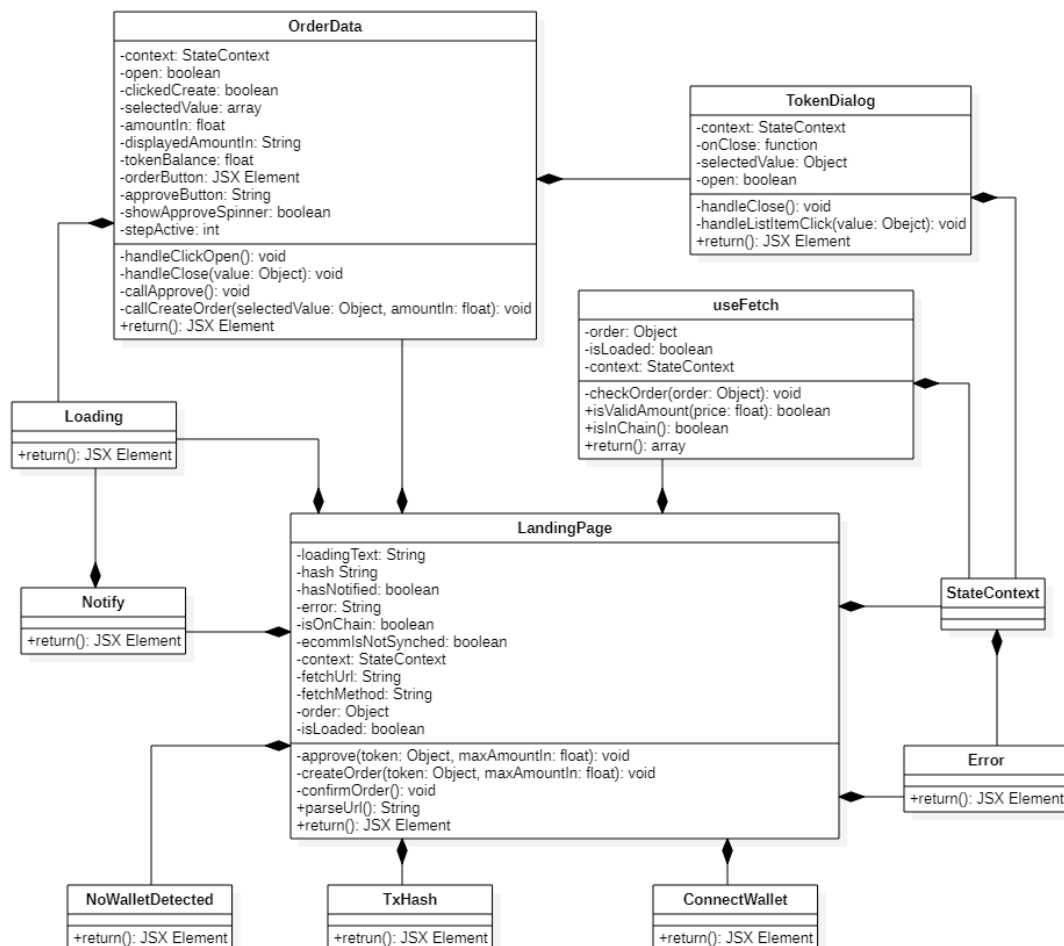
1. La DApp chiama la *_orderOperation(id, expr, orderAmount)* dall'oggetto StateContext(rappresentato nel codice da context) usando i parametri necessari per la Refund.
2. Viene ricevuto il risultato della funzione *askRefund(id)* e in base a ciò si aggiornano le informazioni (balance e stato ordine) o si riceve un messaggio di errore.

2.3.3 Diagramma di Attività: Accesso HomePage



2.4 Landing Page

2.4.1 Diagramma delle classi



Il diagramma delle classi della Landing Page è basato sull'omonimo componente React **Landing Page**, che fa da padre alla gerarchia di questa interfaccia utente. In questo caso però **Landing Page** renderizzerà solo alcuni componenti alla volta in base alla situazione in cui ci troviamo:

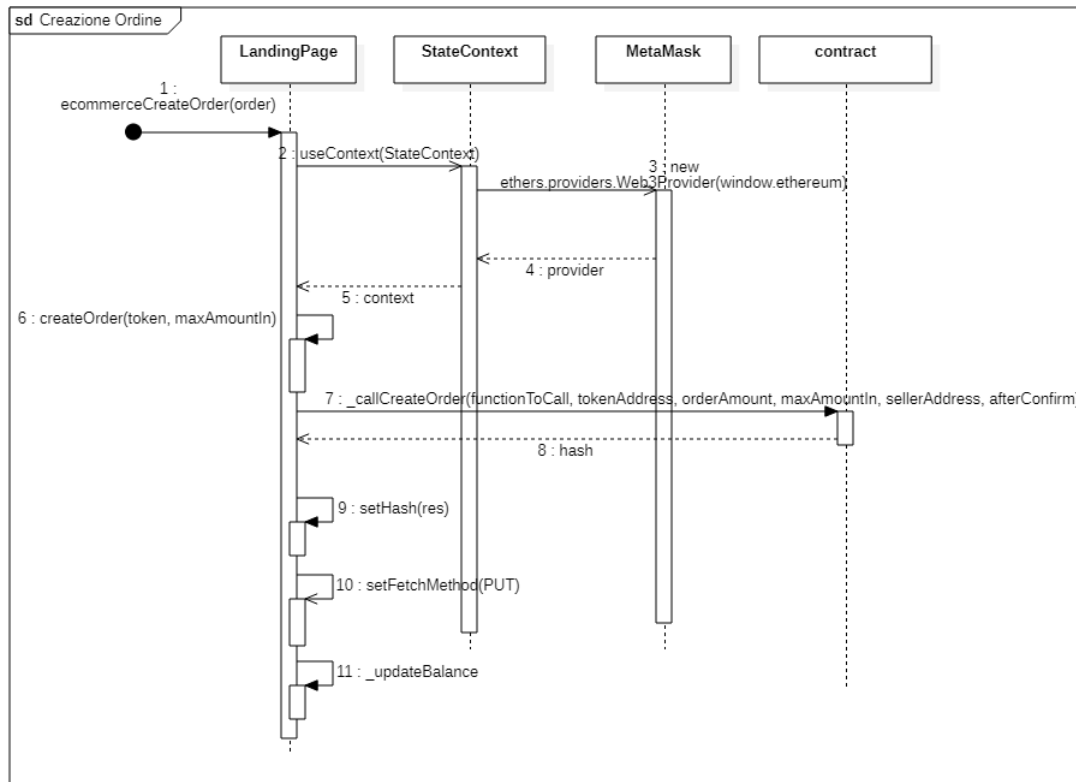
- appena arrivati sulla Landing Page verrà visualizzato il componente **OrderData** in cui vengono visualizzati i dati della transazione e il componente **TokenDialog** dove l'utente può selezionare il token con cui preferisce pagare;
- dopo aver eseguito la transazione verrà renderizzato il componente **Notify** per avvisare l'utente che l'e-commerce è stato avvisato correttamente dell'avvenuto pagamento e quando la transazione sarà minata in blockchain verrà renderizzato il componente **TxHash** che mostrerà all'utente l'hash della transazione appena minata.

UseFetch è un "Custom Hook" realizzato per prendere le informazioni del pagamento dal server dell'e-commerce quando si accede alla Landing Page e successivamente notificarlo

del pagamento avvenuto correttamente. Anche qui sono presenti i componenti **NoWallet-Detected**, **SwitchNetwork**, **ConnectWallet** e **StateContext** già descritti in precedenza.

2.4.2 Diagrammi di sequenza

2.4.2.1 Creazione dell'ordine



Il diagramma descrive i passi per la creazione di un ordine:

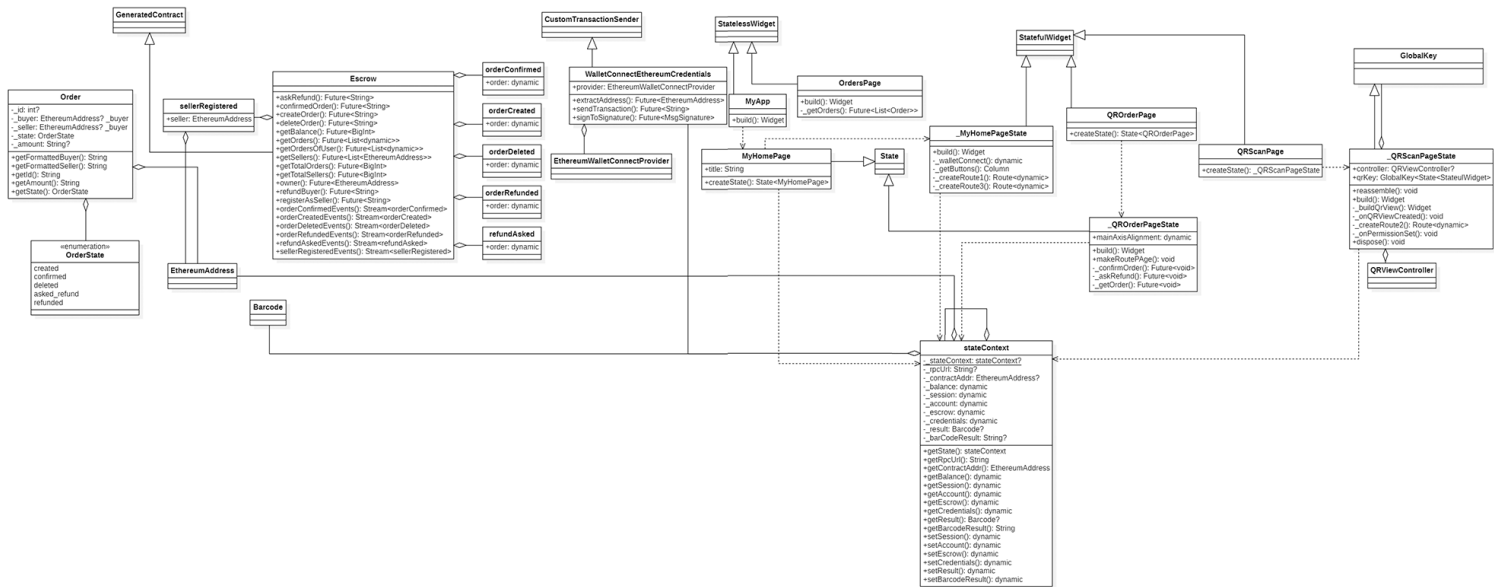
1. La richiesta dell'ordine inizia dall'e-commerce dove il cliente acquista il prodotto e lo indirizza alla nostra LandingPage.
2. Dalle chiamate 2 a 5 avviene la connessione del Wallet del cliente e vengono forniti i dati necessari alla landing page attraverso un oggetto context.
3. Dalle chiamate 6 a 8 avviene la creazione dell'ordine vero e proprio chiamando il metodo di contract `_callCreateOrder(functionToCall, tokenAddress, orderAmount, maxAmountIn, sellerAddress, afterConfirm)`.
4. Infine dopo la ricezione di hash per vedere se la transazione è avvenuta con successo, si aggiorna il balance dell'utente.

2.5 App Mobile

2.5.1 Diagramma delle classi

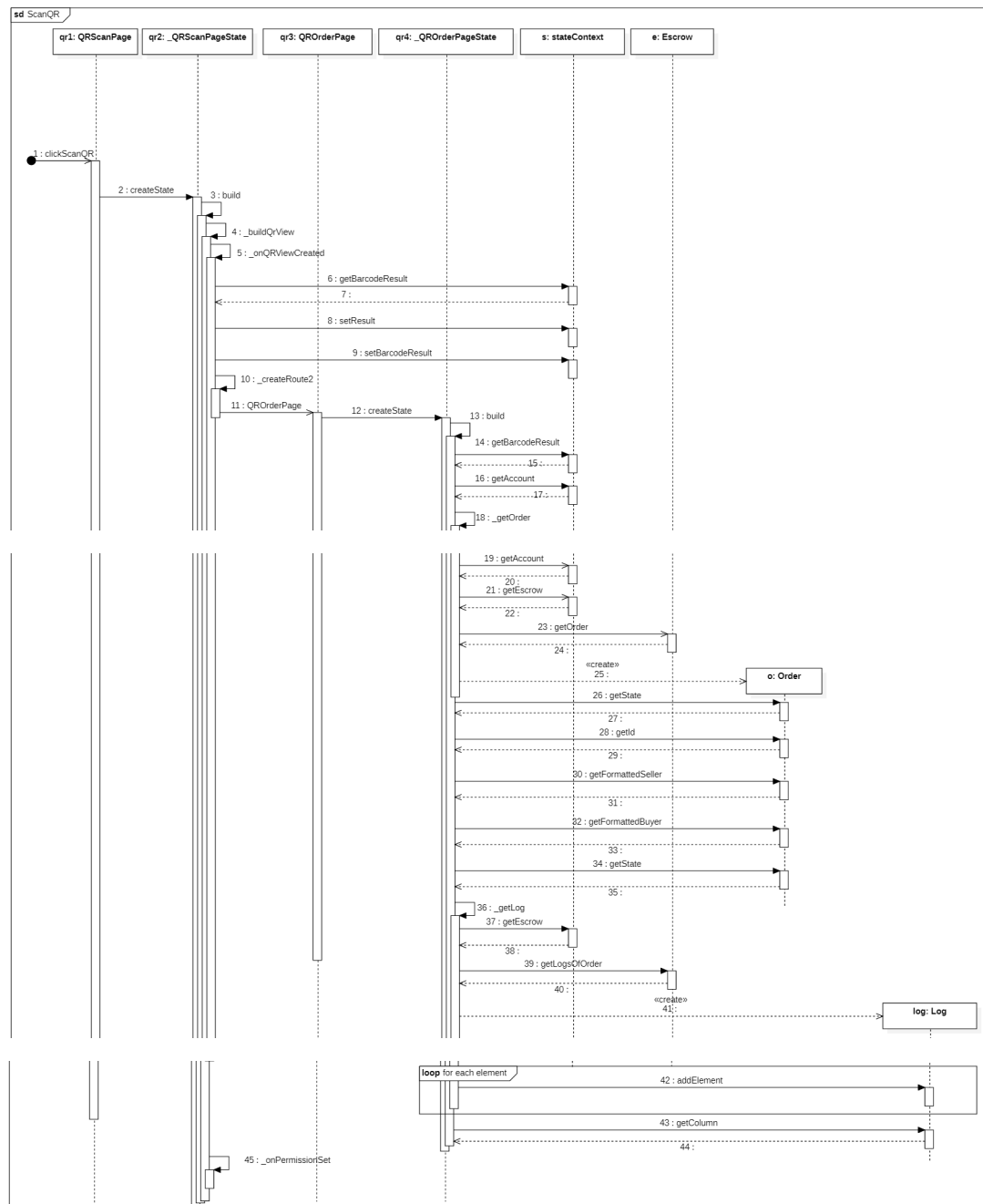
Il diagramma delle classi dell'applicazione mobile si basa sulla classe **_MyHomePageState**, la quale gestisce la connessione al wallet dell'utente e il routing verso le viste dell'app, quali la pagina di scansione del QR code e la visualizzazione degli ordini. Le viste principali sono implementate dalle classi **QROrderPage**, per la visualizzazione dei dettagli dell'ordine dopo aver scannerizzato un ordine, e **OrdersPage**, per la renderizzazione della lista degli ordini relativi al wallet connesso.

In questo diagramma è stato implementato il design pattern *Singleton* nella classe **state-Context** per fare in modo che ci sia una sola istanza dello smart contract operativa all'interno dell'applicazione.



2.5.2 Diagrammi di sequenza

2.5.2.1 Scannerizzazione QRCode

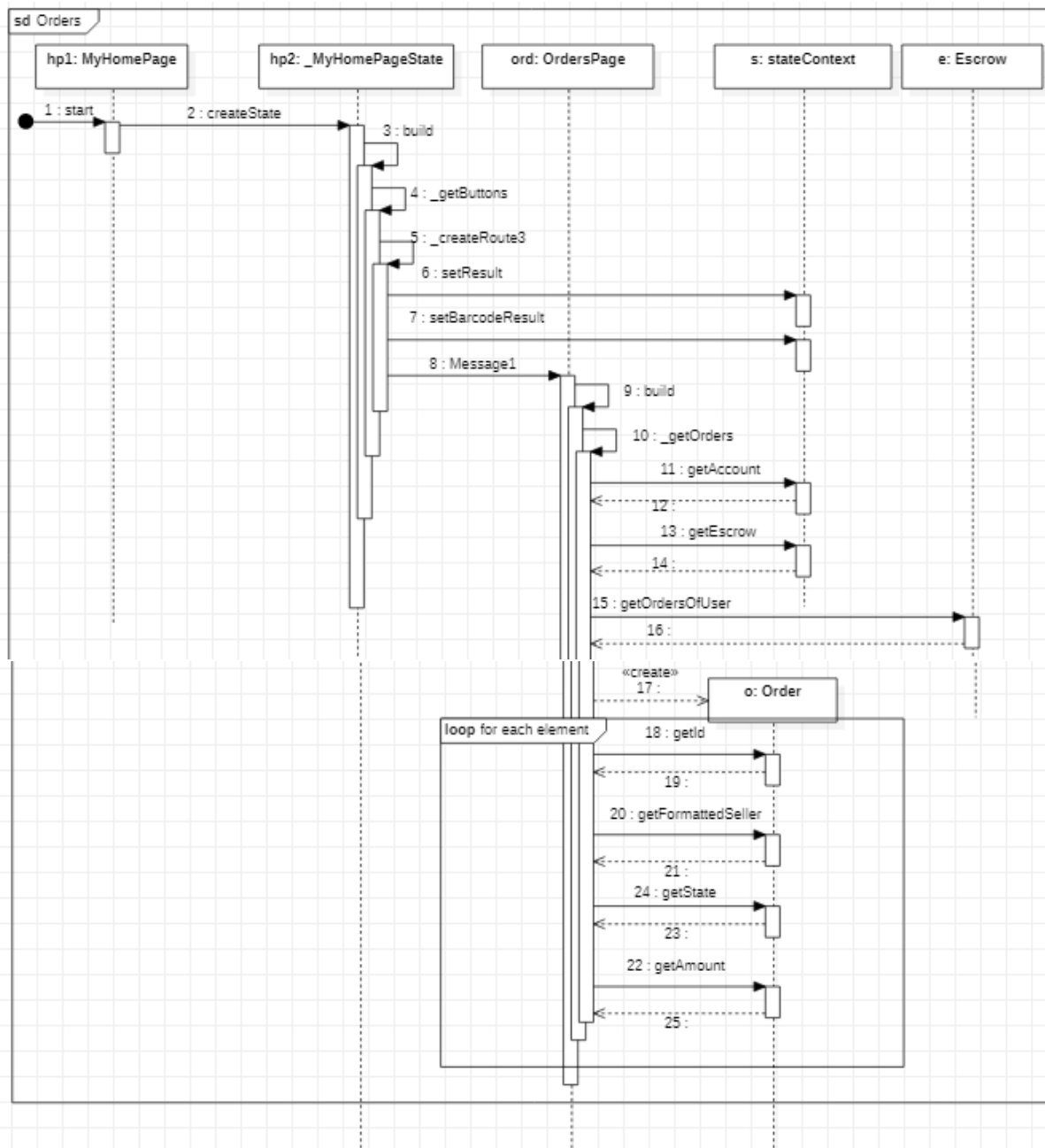


Il diagramma descrive i passi per la scannerizzazione del QRCode:

1. Viene cliccato il button per la scannerizzazione del QRCode, quindi viene creata la pagina QRScanPage e i widget annessi.

2. Quando l'utente inquadra un QR valido il suo contenuto viene trasferito nello state-Context e viene aperta la pagina `QROrderPage` nella quale sono contenuti i dettagli dell'ordine.
3. Nella funzione `build()` di `QROrderPage` infatti viene estrapolato l'id dell'ordine e viene interrogato lo smart contract per ottenere le informazioni a riguardo (`getOrder`) e utilizzarle nella costruzione di un nuovo oggetto della classe `Order`.
4. Infine un `FutureBuilder` (il quale in attesa dell'arrivo dell'ordine) prepara i widget necessari alla sua visualizzazione sulla pagina dell'applicazione.

2.5.2.2 Visualizzazione ordini del buyer



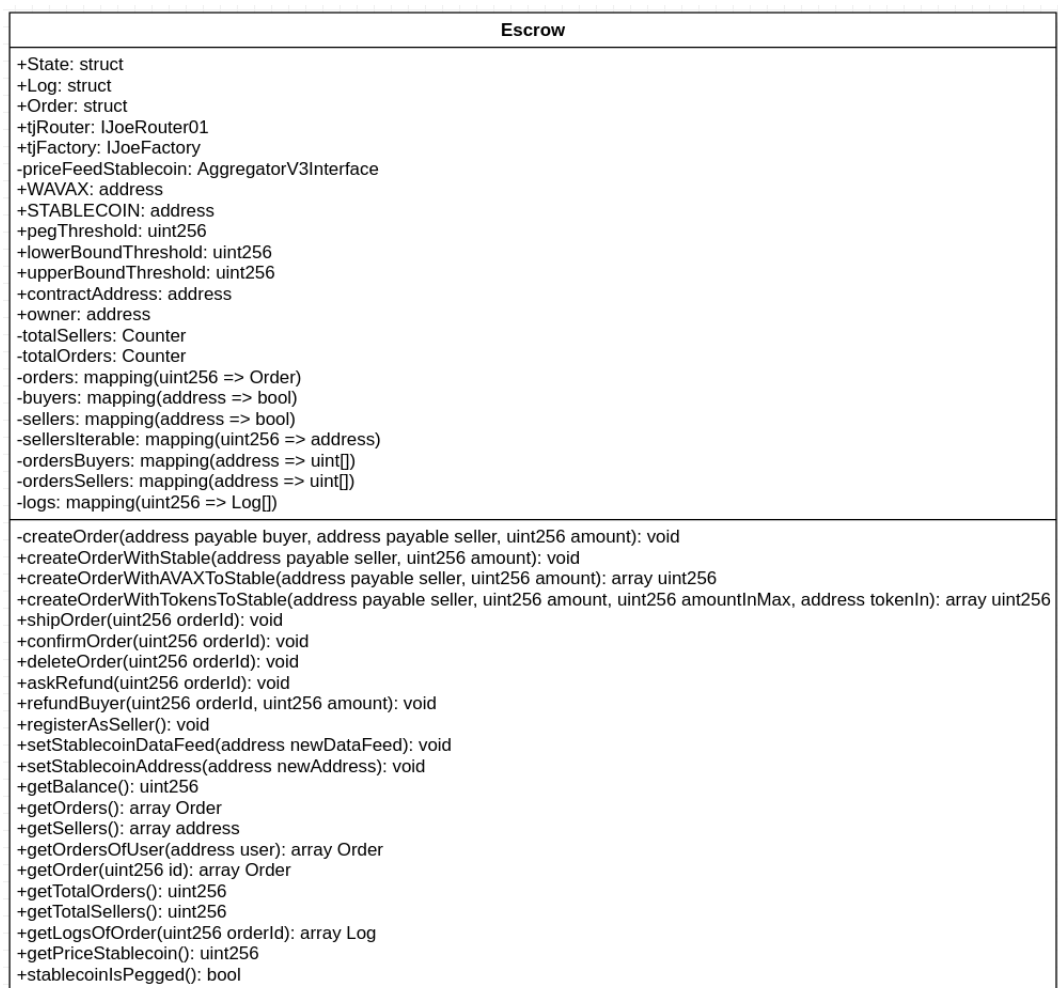
Il diagramma descrive i passi per la visualizzazione dei ordini del buyer:

1. le chiamate da 1 a 9 costruiscono i widget necessari e inizializzano lo stato necessario per la visualizzazione dell'ordine.
2. Quindi viene chiamata `_getOrders` che fa una chiamata allo `stateContext` per acquisire account ed escrow su cui chiamare `getOrdersOfUser`(metodo dello SmartContract).

3. Per ogni order vengono create istanze della classe Order (con ID, address del seller, stato e amount speso per l'ordine).
4. Un FutureBuilder in attesa del completamento di questa chiamata predispone la tabella basandosi sugli oggetti Order ritornati da `_getOrders`.

2.6 Smart Contract

2.6.1 Diagramma delle classi



Lo smart contract è la parte che si occupa della business logic e di salvare tutti i dati (ordini, compratori, venditori...) on-chain, tramite apposite strutture (**Order**, **State**, **Log** e **mappings**).

La maggior parte dei metodi è invocabile solamente da specifici attori, grazie all'uso dei **modifiers**. Ad esempio, il metodo **confirmOrder(orderId)** è invocabile solamente dal compratore che possiede quell'ordine specifico, i **setters** sono invocabili solamente dall'owner dello smart contract e i **getter** da chiunque.

Negli attributi si possono trovare **tjRouter** e **tjFactory**, delle interfacce che permetteranno al nostro smart contract di interagire con altri due smart contract della piattaforma Trader Joe su Avalanche (e nella testnet Rinkeby), il DEX sfruttato per la conversione in stablecoin.

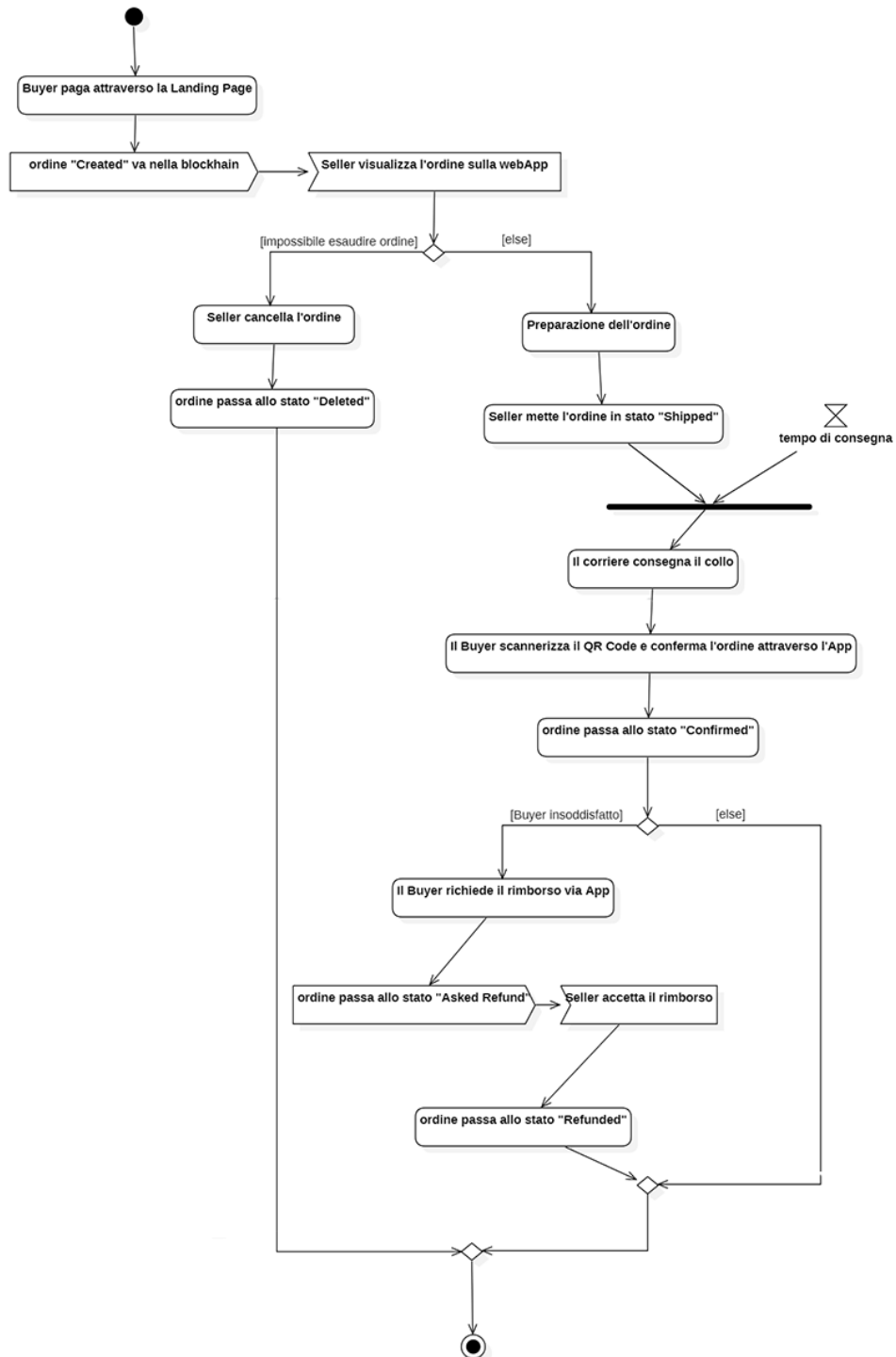
A seconda del token scelto dal compratore per il pagamento, viene invocato dalla webApp uno dei tre metodi **createOrderWith***, ciascuno dei quali invoca il metodo principale **createOrder** che va effettivamente a salvare i dati nei mappings.

Ogni volta che viene invocato uno dei due metodi che interagiscono con il router di Trader Joe (**createOrderWithAVAXtoStable** e **createOrderWithTokensToStable**), viene controllato il prezzo della stablecoin scelta. Se esce dal `pegThreshold` prestabilito, il pagamento viene bloccato per motivi di sicurezza (il venditore andrebbe in perdita).

Per lo scambio in stablecoin abbiamo deciso di utilizzare i metodi di tipo "**ExactOut**" predisposti dal router, e non "**ExactIn**". Questo per evitare altri controlli di sicurezza che, con il primo tipo di funzione, risultano superflui dato che se alla fine della conversione il chiamante non ottiene **esattamente** il numero di token in output richiesti, tutta la transazione viene annullata.

La tipologia "**ExactIn**" invece espone il chiamante a problemi di sicurezza quali "frontrunning sandwich" o altri metodi di manipolazione del prezzo. Per altre info, leggere la documentazione di [Uniswap](#) (TraderJoe è un fork di UniswapV2, la documentazione quindi coincide).

2.6.2 Diagramma di Attività: Ciclo di vita di un ordine



3 Possibili punti di estensione

Nelle sezioni successive sono state descritte eventuali funzioni implementabili per l'estensione e la manutenzione di ShopChain. Tali funzionalità sono state progettate per coprire diversi aspetti del prodotto, tra cui:

1. favorire un'esperienza utente migliore
2. incentivare il comportamento corretto degli utenti, sia per i Seller che per i Buyer

3.1 Fees rimborsate dall'e-commerce

Nello stato attuale, il sistema prevede che il Buyer paghi le fees necessarie per l'uso di servizi aggiuntivi (ad esempio la conversione di stablecoin), dato che ricade su quest'ultimo la scelta di utilizzarli o meno.

Dopo un periodo di confronto di idee, si è pensato che fosse comunque opportuno fornire la possibilità al Seller di decidere o meno di rimborsare le fees ai Buyer. Questo può risultare utile al Seller, che può sfruttare campagne pubblicitarie dove può con ulteriori policy rendere il proprio e-commerce più competitivo.

Passi necessari:

1. Modificare la funzione **registerSeller()** in modo da impostare un flag per il rimborso delle fees;
2. Modificare appositamente i metodi invocabili dai Buyer;
3. Periodicamente il Seller dovrà fornire la liquidità necessaria allo smart contract per effettuare il rimborso;

3.2 Ottimizzazione filtri

La funzionalità di filtraggio per gli ordini, in base allo stato o all'address del wallet interessato, è a carico del lato front-end della webApp.

In casi estremi (ad es: grande quantità di record) si possono incontrare problemi riguardanti la performance, danneggiando quindi di fatto l'esperienza dell'utente. Per questa ragione si è ipotizzato di usare theGraph, un protocollo di indicizzazione per blockchain come Ethereum e Avalanche, spostando la complessità del filtraggio dal front-end alla blockchain.

3.3 Sistema di reward

Si è considerata l'adozione di un sistema di reward affinché l'utente sia incentivato a pagare a rate utilizzando un token ERC-20, creato appositamente per la piattaforma, durante fasi diverse della consegna.

Questo può essere realizzato creando nuovi stati intermedi durante la spedizione, in modo tale da minimizzare il rischio che l'utente si comporti in modo scorretto.

Un'alternativa è quella di ricompensare l'utente dopo la scansione del QR Code, sempre con il token della piattaforma.

3.4 Verifica seller reali

La registrazione dei Seller non richiede particolari requisiti, e per questo crea una potenziale vulnerabilità, ossia possibili spam di registrazioni usando profili fittizi.

Per questa ragione il team ha pensato di creare in futuro un sistema di verifica basato su un deposito momentaneo di token (es: 1 AVAX) che poi verrà restituito al Seller dopo un determinato numero di ordini confermati.

3.5 Strategie di investimento per i fondi bloccati

Quando viene creato un ordine viene depositata liquidità nello smart contract che però rimane inutilizzabile durante tutta la fase di preparazione dell'ordine, fino alla consegna e allo sblocco tramite QR Code (o eventuale refund/delete). I fondi in questo modo diventano inutilizzabili da parte del compratore e potrebbe venire meno l'opportunità cost della transazione in sé.

Per ovviare a questa problematica si potrebbe dare la possibilità ai Buyer di scegliere strategie di investimento per trarre vantaggio dai propri token anche quando sono bloccati, ad esempio creando una specie di farming-pool che frutterà ricompense sottoforma di token della piattaforma, oppure usare i vault strategici di [Yearn Finance](#).

4 Funzioni nella codifica

4.1 Smart Contract

```
function createOrder(address payable buyer, address payable seller,  
uint256 amount) internal:
```

Metodo "internal" (non chiamabile dall'esterno) che va a salvare nello storage i dati dell'ordine.

```
function createOrderWithStable(address payable seller, uint256 amount) external  
payable sellerExists(seller):
```

Invocato se l'utente vuole pagare con la stablecoin della piattaforma. Prima di chiamarlo, l'utente deve richiamare il metodo "approve" dell'interfaccia IERC-20 della stablecoin apposita, per dare il permesso (allowance) allo smart contract di prelevare i fondi dal proprio indirizzo. Richiama il metodo "createOrder".

```
function createOrderWithAVAXToStable(address payable seller, uint256 amount)  
external payable sellerExists(seller) peggedStablecoin returns (uint256[]  
memory amounts):
```

Invocato se l'utente vuole pagare in AVAX e quindi, prima di richiamare "createOrder", viene effettuato lo scambio in stablecoin con il metodo "swapAVAXForExactTokens" del router di Trader Joe, e al termine dell'operazione rimarrà l'amount equivalente in stablecoin dentro allo smart contract.

```
function createOrderWithTokensToStable(address payable seller, uint256 amount,  
uint256 amountInMax, address tokenIn) external payable sellerExists(seller)  
peggedStablecoin returns (uint256[] memory amounts):
```

Invocato se l'utente vuole pagare con un qualsiasi token ERC-20; il comportamento è simile al metodo soprastante ma viene chiamato "swapTokensForExactTokens", e occorre garantire il permesso tramite "approve", come per il metodo "createOrderWithStable".

```
shipOrder(uint256 orderId) external orderExists(orderId)  
sellerIsOwner(orderId):
```

Invocabile solo dal Seller per cambiare lo stato dell'ordine in "Shipped" dallo stato "Created".

```
confirmOrder(uint256 orderId) external orderExists(orderId)  
buyerIsOwner(orderId):
```

Invocabile solo dal Buyer dall appMobile al momento della scansione del QR Code. Il denaro viene sbloccato e mandato dallo smart contract al Seller.

```
deleteOrder(uint256 orderId) external orderExists(orderId)  
sellerIsOwner(orderId):
```

Invocabile solo dal Seller se l'ordine si trova nello stato "Created" o "Shipped". Il denaro viene inviato dallo smart contract al Buyer

`askRefund(uint256 orderId) external orderExists(orderId) buyerIsOwner(orderId) :`
Invocabile solo dal Buyer se l'ordine si trova nello stato "Created", "Shipped" o "Confirmed".

`refundBuyer(uint256 orderId, uint256 amount) external payable`
`orderExists(orderId) sellerIsOwner(orderId) :`
Invocabile dal Seller solo se il Buyer ha chiesto il rimborso. Il Seller invia il denaro al Buyer.

`registerAsSeller() :`
Chiamato da chiunque voglia registrarsi come Seller all'interno della piattaforma.

`setStablecoinDataFeed(address newDataFeed) external onlyOwner :`
Invocabile solo dall'owner del contratto, serve a cambiare il DataFeed di Chainlink, ossia l'oracolo tramite il quale si può richiedere il prezzo corrente della stablecoin monitorata.

`setStablecoinAddress(address newAddress) external onlyOwner :`
Invocabile solo dall'owner del contratto, serve a cambiare l'address della stablecoin scelta per le conversioni.

`setStablecoinPegThreshold(uint256 newThreshold) external onlyOwner :`
Invocabile solo dall'owner del contratto, serve a settare un nuovo threshold, ossia un bound oltre al quale il prezzo della stablecoin non può scendere/salire.

4.2 WebApp

DApp.jsx

Descrizione: classe che si occupa di gestire la logica di renderizzazione dei componenti che formano la webApp, oltre ad inizializzare i dati dell'utente e della blockchain.

`async _setListenerMetamaksAccount() :`
Listener che si occupa di gestire il cambio dell'account con cui si è collegati alla webApp.

`async _setListenerNetworkChanged() :`
Listener che si occupa di avvisare l'utente sulla giusta blockchain se si trova sulla giusta blockchain.

`async _initialize() :`
Funzione che si occupa di inizializzare i dati dello smart contract e della blockchain.

`async _changeNetwork() :`
Funzione che si occupa di indirizzare l'utente sulla giusta blockchain nel caso in cui sia collegato ad una blockchain diversa da quella su cui opera la webApp.

`async _loadBlockchainData() :`
Funzione che si occupa di inizializzare i dati dalla blockchain, quali gli ordini e il balance dello smart contract.

`async _refreshInfo(tx) :`

Funzione che si occupa di aggiornare le informazioni visibili dall'utente dopo ogni interazione di quest'ultimo con le funzioni dello smart contract. `tx` rappresenta la transazione che viene creata ogni volta che l'utente interagisce con lo smart contract.

```
async _initializeOrders() :
```

Funzione che recupera la lista degli ordini relativi ad un utente salvati nello smart contract.

```
async _orderOperation(id, expr, orderAmount) :
```

Funzione che gestisce le operazioni sull'ordine eseguite dall'utente identificato da un `id`; `expr` indica l'operazione da eseguire sull'ordine e `orderAmount` serve per confermare la richiesta di reso.

```
async _getContractBalance() :
```

Funzione che si occupa di recuperare il balance all'interno dello smart contract.

Orders.jsx

Descrizione: componente che si occupa di renderizzare la lista degli ordini dell'utente e applicare dei filtri a quest'ultima.

```
applyFilters() :
```

Funzione che viene invocata al click del corrispondente bottone e che applica i filtri inseriti dall'utente.

```
filterOrdersByAddress(address) :
```

Funzione che filtra la lista degli ordini per `address` preso in input.

```
filterOrdersByState(ordersToFilter, state) :
```

Funzione che filtra la lista degli ordini `ordersToFilter` per `state` preso in input.

```
copyAddress(address, id) :
```

Permette di copiare un indirizzo `address` compratore o venditore di un ordine identificato da `id`.

```
visualizeOrder(order) :
```

Funzione che si occupa di creare una riga di una tabella contenente le informazioni di `order` preso in input.

RegisterSeller.jsx

Descrizione: componente che permette di registrare nello smart contract un indirizzo come venditore.

```
async _registerSeller() :
```

Funzione che registra il wallet collegato alla webApp come venditore.

StateContext.jsx

Descrizione: classe che contiene tutti i metodi che vengono utilizzati da più componenti.

`async _connectWallet() :`

Funzione che inizializza lo smart contract e i dati più importanti della webApp dalla blockchain.

`async _changeNetwork(networkName) :`

Gestisce il cambio blockchain nel caso in cui `networkName` sia la rete sbagliata.

`async _wrongChain() :`

Setta il campo `rightChain` a *false* se l'utente è collegato alla blockchain sbagliata.

`async _rightChain() :`

Setta il campo `rightChain` a *true* se l'utente è collegato alla blockchain giusta.

`_setListenerMetamaksAccount() :`

Listener che gestisce il cambio dell'account collegato alla webApp.

`_setListenerNetworkChanged() :`

Listener che gestisce il cambio della blockchain a cui è collegato l'utente.

`async _updateBalance() :`

Aggiorna il balance dell'utente dopo ogni interazione con lo smart contract.

`async _getOrderById(id) :`

Ritorna un ordine identificato da `id` se esiste, altrimenti non ritorna nulla.

`async _callCreateOrder(functionToCall, tokenAddress, orderAmount, maxAmountIn, sellerAddress, afterConfirm) :`

Chiama tre diversi metodi dello smart contract in base al metodo di pagamento selezionato dall'utente nella Landing Page rappresentato da `functionToCall`.

`async _orderOperation(id, expr, orderAmount) :`

Chiama le diverse operazioni da eseguire sull'ordine identificato da `id` in base alla funzione che riceve in input rappresentata da `expr`.

`async _getSellers() :`

Ritorna un array contenente gli indirizzi dei venditori salvati nello smart contract.

`async _userIsSeller() :`

Controlla se l'indirizzo dell'utente collegato è un venditore oppure no e di conseguenza setta il campo `userIsSeller`.

`_isAuthorizedSeller(sellerAddress) :`

Controlla se `sellerAddress` è effettivamente un venditore o meno.

```
async _getQRCode(order) :
```

Crea il QR code di `order` necessario al venditore.

```
async _getLog(id) :
```

Ritorna il log di cambio stato di un ordine identificato da `id`.

```
async _getERC20Balance(token) :
```

Ritorna il balance dell'indirizzo collegato alla Landing Page convertito in base a `token` preso in input.

```
async _approveERC20(tokenAddress, amount) :
```

Funzione che consente allo smart contract rappresentato da `tokenAddress` di prendere una quantità di token pari ad `amount` dal wallet dell'utente per convertirli.

```
async _ERC20isApproved(tokenAddress, amount) :
```

Controlla se un utente ha già eseguito l'approve su un determinato smart contract rappresentato da `tokenAddress`.

```
_setOrderStateChangedFalse() :
```

Setta il campo `orderStateChanged` a *false*.

```
aync _getAmountsIn(token, amountOut) :
```

Coverte `amountOut` nel corrispettivo in `token`.

4.3 Landing Page

LandingPage.jsx

```
async createOrder(token, maxAmountIn) :
```

Gestisce la creazione dell'ordine in base a `token` preso in input: se la transazione va a buon fine viene visualizzato l'hash e viene notificato l'e-commerce dell'avvenuto pagamento, altrimenti viene visualizzato un messaggio di errore.

```
parseUrl() :
```

Ritorna l'url da cui prendere le informazioni dell'ordine con cui creare la transazione.

OrderData.jsx

```
async callApprove() :
```

Funzione che chiama `async _approveERC20(tokenAddress, amount)` di StateContext e che mostra all'utente il risultato della funzione di approve.

useFetch.jsx

```
isValidAmount(price) :
```

Controlla se `price` è un numero decimale più grande di 0.

```
checkOrder(order) :
```

Controlla che `order` sia un ordine valido di cui non sia già stata creata la transazione corrispettiva.

4.4 App Mobile

Order.dart

Descrizione: classe che offre una rappresentazione dart del tipico ordine effettuato con ShopChain, essa semplifica il codice e facilita il testing.

```
Order(this._id, this._buyer, this._seller, String state, this._amount) :
```

Semplice costruttore che assegna i vari valori degli attributi della classe, converte la stringa state in un oggetto enum OrderState.

```
String getFormattedBuyer() :
```

Metodo pubblico che aiuta la visualizzazione dei lunghi indirizzi dei vari utenti utilizzati nelle blockchain.

```
String getFormattedSeller() : Simile al precedente.
```

```
int getId() :
```

Getter della variabile privata `_id`.

```
String getAmount() :
```

Getter della variabile privata `_amount`.

```
int getState() :
```

Getter della variabile privata `_state`.

OrderState.dart

Descrizione: enumerazione utilizzata per rappresentare su flutter gli stati in cui si può trovare un ordine.

StateContext.dart

Descrizione: classe di tipo singleton che raccoglie diverse informazione la cui fruibilità era necessaria a diverse pagine.

```
static stateContext getState() :
```

Restituisce l'istanza preesistente di stateContext o ne costruisce una nuova.

`String getRpcUrl() :`

Getter del 'Remote Procedure Call' link, viene utilizzato durante la connessione al wallet.

`EthereumAddress getContractAddr() :`

Getter dell'indirizzo sul quale è stato deployato lo smart contract di ShopChain.

`dynamic getBalance() :`

Getter del bilancio dello smart contract.

`dynamic setSession(session) :`

La variabile `_session` viene settata durante la connessione al wallet.

`dynamic getSession() :`

Getter della variabile `_session`.

`dynamic setAccount(account) :`

Memorizzazione dell'account con il quale è stato effettuato l'accesso a MetaMask per essere successivamente utilizzato nell'app.

`dynamic getAccount() :`

Getter della variabile `_account`.

`dynamic setEscrow(escrow) :`

Memorizzazione della variabile che rappresenta lo smart contract e del quale è una finestra d'accesso alla quale inoltrare le chiamate ai metodi richiesti dall'utente.

`dynamic getEscrow() :`

Getter della variabile `_escrow`.

`dynamic setCredentials(credentials) :`

Memorizzazione delle `EthereumCredentials` dell'utente.

`dynamic getCredentials() :`

Getter della variabile `_credentials`.

`dynamic setResult(result) :`

Setter della variabile contenente l'oggetto `BarCode` letto dalla fotocamera sulla pagina `QR-ScanPage`.

`BarCode? getResult() :`

Getter della variabile `_result`.

`dynamic setBarcodeResult(result) :`

Setter della stringa contenuta nel codice QR scannerizzato sulla pagina `QRScanPage`.

`String? getBarcodeResult() :`

Getter della variabile `_barCodeResult`.

MyHomePage.dart

Descrizione: questa classe è una derivazione della classe astratta `StatefulWidget`, il suo compito è predisporre i widget della vista principale (presentata all'avvio dell'applicazione).

`Widget build(BuildContext context) :`

Principale funzione di costruzione di ogni widget flutter, predispone i componenti di layout della pagina.

`Column_getButtons() :`

Lo scopo di questa funzione consiste nel presentare all'utente un set di bottoni differente in base alla connessione al wallet metamask. Una volta connesso infatti, le opzioni ScanQR e Orders vengono presentate.

`void _walletConnect() async :`

Questo metodo utilizza la libreria `walletconnect_dart` e si occupa della comunicazione con il wallet installato sul dispositivo utente affinché gli sia richiesto l'accesso.

`Route _createRoute1() :`

Funzione di passaggio dalla pagina principale a `QRScanPage`.

`Route _createRoute3() :`

Funzione di passaggio dalla pagina principale a `OrdersPage`.

OrdersPage.dart

Descrizione: classe derivata da `StatelessWidget` che predispone e regola il comportamento dei widget che compongono la pagina rappresentante la tabella degli ordini dell'utente.

`Widget build(BuildContext context) :`

Principale funzione di costruzione di ogni widget flutter, predispone i componenti di layout della pagina.

`Future<List<Order> _getOrders() :`

Metodo il cui codice richiede una chiamata allo smart contract la cui risposta contiene gli ordini dell'utente che ha effettuato l'accesso e compone una lista di oggetti della classe `Order` che verranno poi visualizzati.

QRScanPage.dart

Descrizione: classe derivata da `StatefulWidget`, rappresenta la pagina per mezzo della quale è possibile scannerizzare il codice QR legato all'ordine.

```
Widget build(BuildContext context) :
```

Principale funzione di costruzione di ogni widget flutter, predispone i componenti di layout della pagina.

```
Widget _buildQRView(BuildContext context) :
```

Creazione dell'oggetto QRView della libreria qr_code_scanner.dart.

```
void _onQRViewCreated(QRViewController controller) :
```

Funzione che si mette in "ascolto" mentre attende che venga inquadrato un codice QR valido per poi salvare il valore letto e procedere alla costruzione della pagina successiva.

```
Route _createRoute2() :
```

Funzione di passaggio dalla pagina QRScanPage a QROrderPage.

QROrderPage.dart

Descrizione: classe derivata da StatefulWidget che costruisce i vari componenti della pagina dedicata all'esposizione dei dettagli riguardanti l'ordine e che espone le operazioni di Ask Refund e Confirm Order.

```
Widget build(BuildContext context) :
```

Principale funzione di costruzione di ogni widget flutter, predispone i componenti di layout della pagina.

```
Future<dynamic> _getOrder(int id) async :
```

Funzione responsabile del recupero dei dati dell'ordine dallo smart contract e della costruzione del corrispondente oggetto Order.

```
Future<Log> _getLog(int id) async :
```

Simile a _getOrder(int id) ma recupera la serie di stati in cui si è trovato l'ordine finora con il timestamp corrispondente, costruisce un oggetto della classe Log.

```
Future<void> _confirmOrder(String orderID) async :
```

Metodo invocato dal bottone Confirm Order il cui compito consiste nella creazione della transazione e successivamente nel lancio dell'app MetaMask.

```
Future<void> _askRefund(String orderID) :
```

Simile a _confirmOrder ma ovviamente l'ordine passerà allo stato Refund Asked in caso di transazione eseguita correttamente.

```
void makeRoutePage(BuildContext context, Widget pageRef) :
```

Funzione che provvede alla pulizia della variabile contenente il codice QR letto in caso di ritorno alla pagina precedente.

SCEscrow.g.dart

Descrizione: questo file .dart è automaticamente generato dalla libreria dart_web3 basandosi sul codice solidity dello smart contract, esso offre una chiamata via dart di tutte le funzioni del file SCEscrow.sol.

```
Future<String> askRefund(BigInt _orderId, required _i1.Credentials credentials,
    _i1.Transaction? transaction);

Future<String> confirmOrder(BigInt _orderId, required _i1.Credentials
    credentials, _i1.Transaction? transaction);

Future<String> createOrder(_i1.EthereumAddress _seller,
    required _i1.Credentials credentials, _i1.Transaction? transaction);

Future<String> deleteOrder(BigInt _orderId, required _i1.Credentials
    credentials, _i1.Transaction? transaction);

Future<BigInt> getBalance(_i1.BlockNum? atBlock);

Future<List<dynamic>> getOrders(_i1.BlockNum? atBlock);

Future<List<dynamic>> getOrdersOfUser(_i1.EthereumAddress _user,
    _i1.BlockNum? atBlock);

Future<List<_i1.EthereumAddress>> getSellers(_i1.BlockNum? atBlock);

Future<BigInt> getTotalOrders(_i1.BlockNum? atBlock);

Future<BigInt> getTotalSellers(_i1.BlockNum? atBlock);

Future<_i1.EthereumAddress> owner(_i1.BlockNum? atBlock);

Future<String> refundBuyer(BigInt _orderId, {required _i1.Credentials
    credentials, _i1.Transaction? transaction});

Future<String> registerAsSeller( {required _i1.Credentials credentials,
    _i1.Transaction? transaction});

Stream<orderConfirmed> orderConfirmedEvents(_i1.BlockNum? fromBlock,
    _i1.BlockNum? toBlock);

Stream<orderCreated> orderCreatedEvents( _i1.BlockNum? fromBlock,
    _i1.BlockNum? toBlock);

Stream<orderRefunded> orderRefundedEvents( _i1.BlockNum? fromBlock,
```

```
_i1.BlockNum? toBlock);  
  
Stream<refundAsked> refundAskedEvents( _i1.BlockNum? fromBlock,  
_i1.BlockNum? toBlock);  
  
Stream<sellerRegistered> sellerRegisteredEvents( {_i1.BlockNum?  
fromBlock, _i1.BlockNum? toBlock});
```

EthereumCredentials.dart

Descrizione: classe che sostiene la creazione delle transazioni necessarie al funzionamento dell'app.

```
Future<EthereumAddress> extractAddress();  
  
Future<MsgSignature> signToSignature(Uint8List payload, int? chainId,  
bool isEIP1559 = false);  
  
Future<String> sendTransaction(Transaction transaction)
```