



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

Indice

1	Registro delle modifiche	2
2	Introduzione	3
2.1	Scopo del documento	3
2.2	Obiettivi del prodotto	3
2.3	Riferimenti	3
2.3.1	Riferimenti informativi	3
2.3.2	Riferimenti tecnici	3
3	Architettura	4
3.1	Pattern architetturale	4
3.2	WebApp	6
3.2.1	Diagramma delle classi	6
3.2.2	Diagrammi di sequenza	7
3.2.2.1	Creazione dell'ordine	7
3.2.2.2	Visualizzazione dei ordini	8
3.2.2.3	Richiesta di rimborso dell'ordine	9
3.3	Mobile	10
3.3.1	Diagramma delle classi	10
3.3.2	Diagrammi di sequenza	11
3.3.2.1	Scannerizzazione QRCode	11
3.3.2.2	Visualizzazione ordini del buyer	13
3.4	SmartContract	14
4	Punti possibili di estensione	15
4.1	Fees rimborsate dall'e-commerce	15
4.2	Ottimizzazione filtri	15
4.3	Sistema di reward	15
4.4	Verifica seller reali	16
4.5	Investimento fondi	16
5	Funzioni nella codifica	17
5.1	SmartContract	17
5.2	WebApp	17
5.3	LandingPage	18
5.4	Mobile	18

1 Registro delle modifiche

v	Data	Nominativo	Ruolo	Descrizione
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 Architettuale §(3.1)
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)

2 Introduzione

2.1 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).

2.2 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.

2.3 Riferimenti

2.3.1 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*

2.3.2 Riferimenti tecnici

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

3 Architettura

3.1 Pattern architetturale

Il team dopo un periodo approfondito di ricerca ha individuato l'architettura più opportuna tra quelle dedicate per le 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 svantaggi che ne conseguono). Questo è possibile se il frontend viene hostato su servizio distribuito come ad esempio IPFS.

Il team ha scelto in particolare questo pattern per questi vantaggi:

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

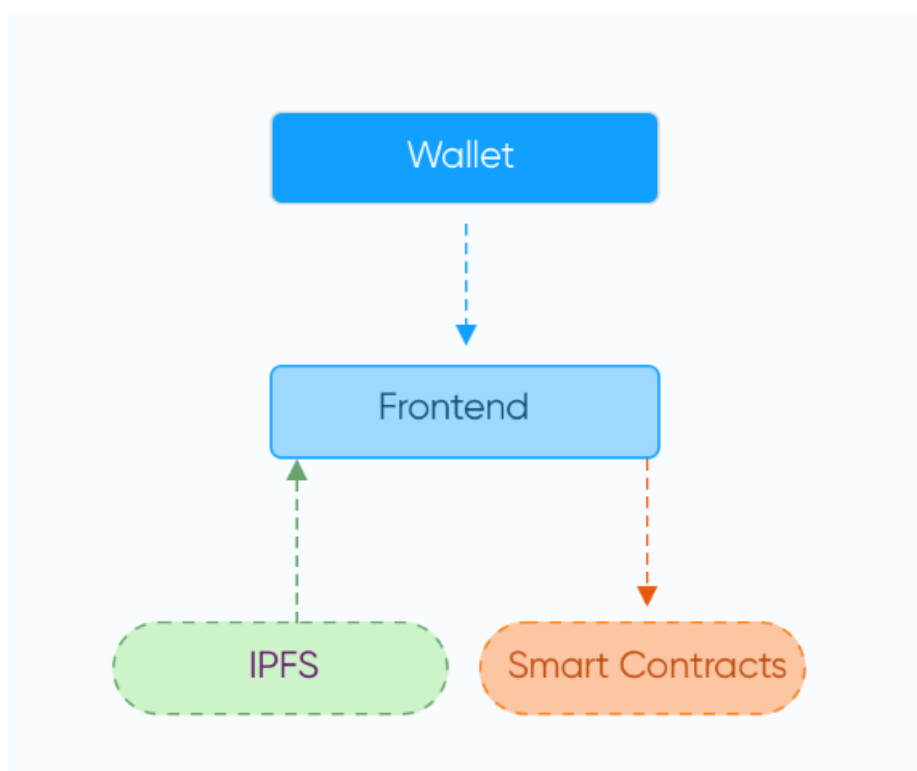


Figura 1: Interazione tra Wallet - Frontend - IPFS - Smart Contract

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 (es. 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.

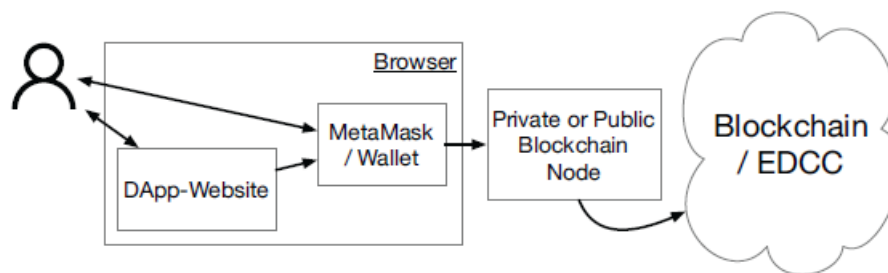


Figura 2: DApp Pattern B - Self-Confirmed Transactions

Fonte:

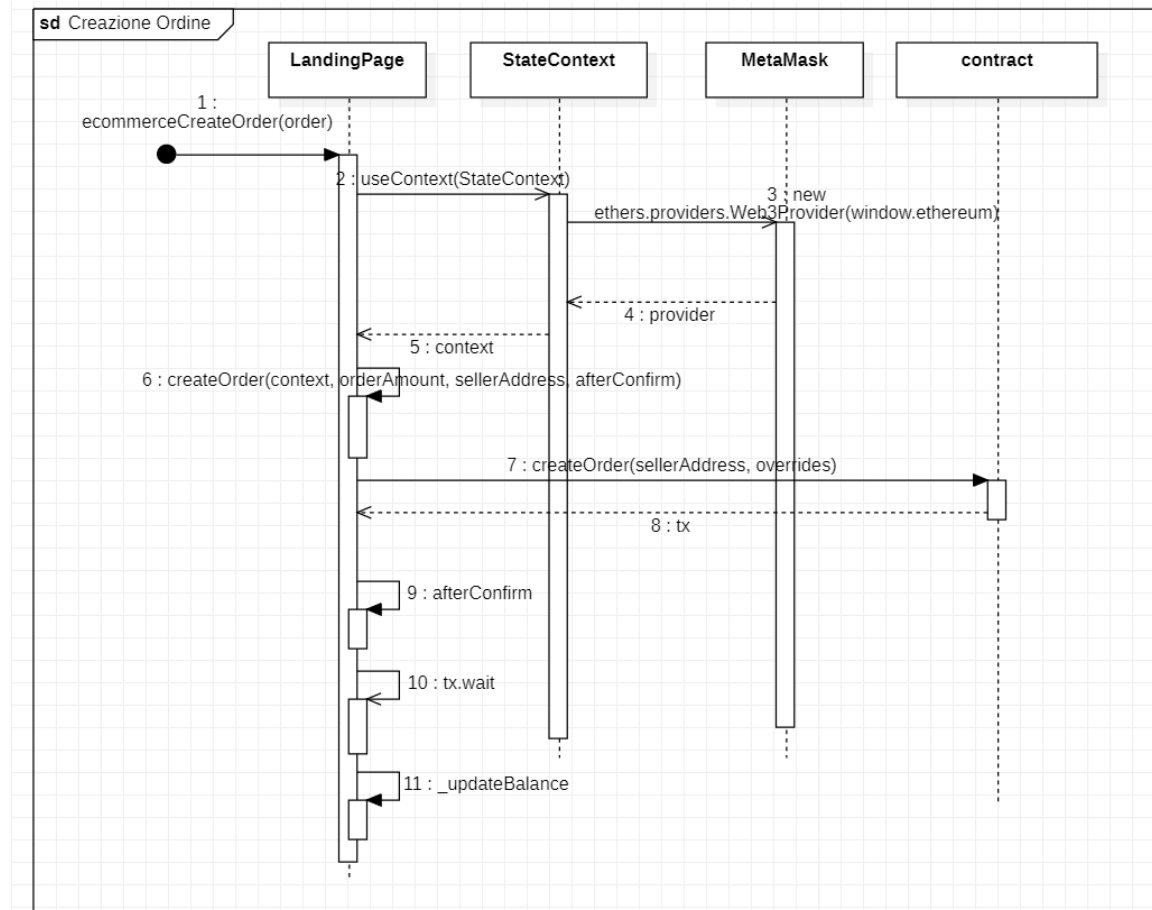
- <https://ieeexplore.ieee.org/abstract/document/8432174>

3.2 WebApp

3.2.1 Diagramma delle classi

3.2.2 Diagrammi di sequenza

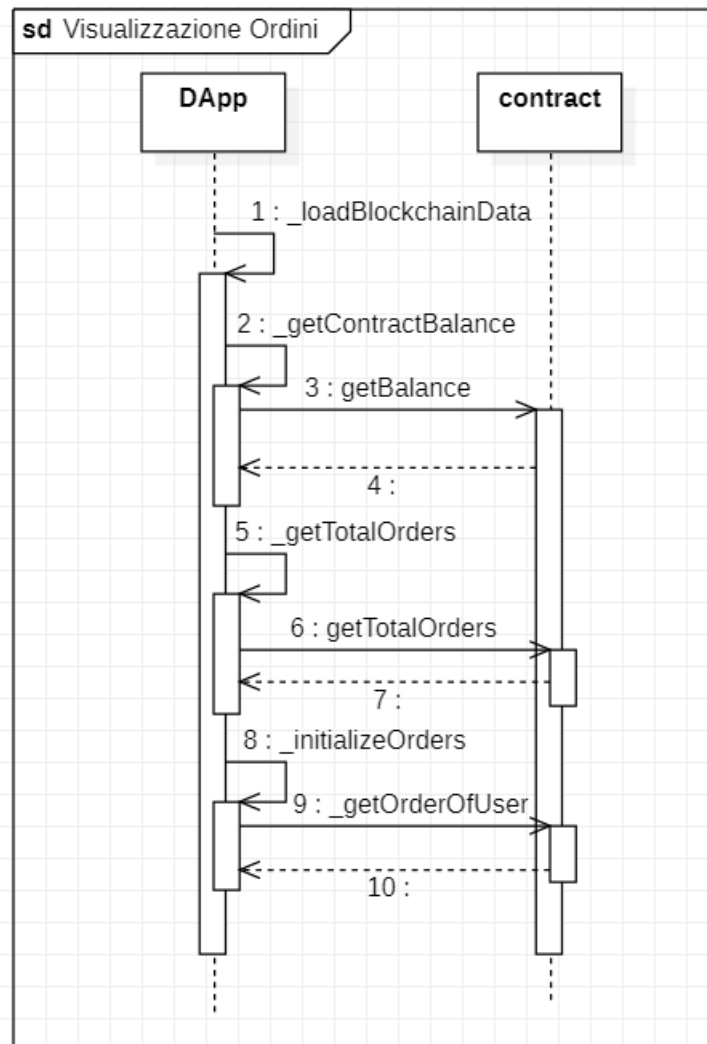
3.2.2.1 Creazione dell'ordine



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 *createOrder(sellerAddress, overrides)*.
4. Infine dopo la ricezione di tx per vedere se la transazione è avvenuto con successo, si aggiorna il balance dell'utente.

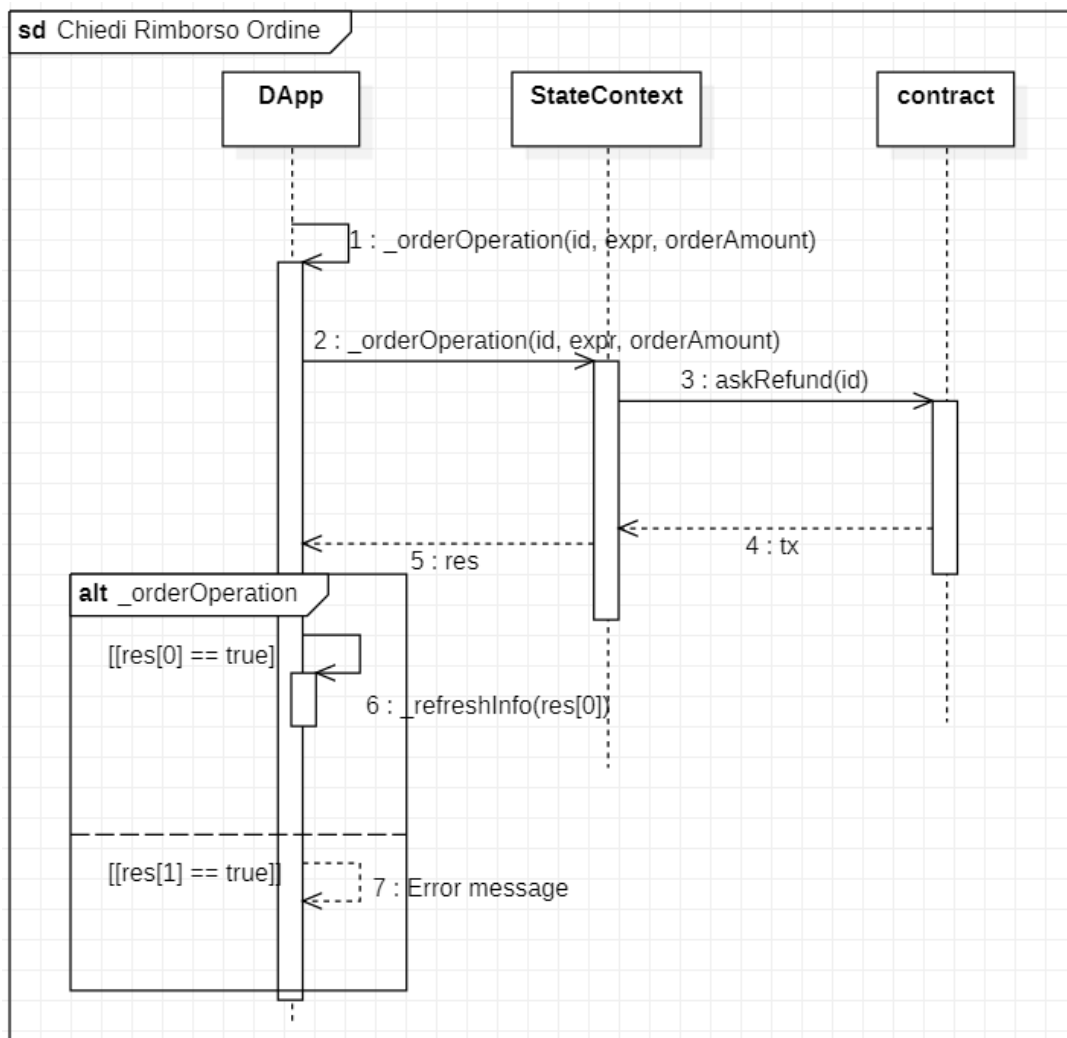
3.2.2.2 Visualizzazione dei ordini



Il diagramma descrive i passi per la visualizzazione dei ordini:

1. Vengono caricati i dati necessari chiamando *_loadBlockchainData()* che è un metodo che incorpora tre getters che chiamano il contract.
2. Viene preso il balance attuale, il numero di ordini totali del contract e gli ordini dell'utente.
3. Ricevuti gli ordini li passerà alla vista Orders che li renderizzerà.

3.2.2.3 Richiesta di rimborso dell'ordine



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

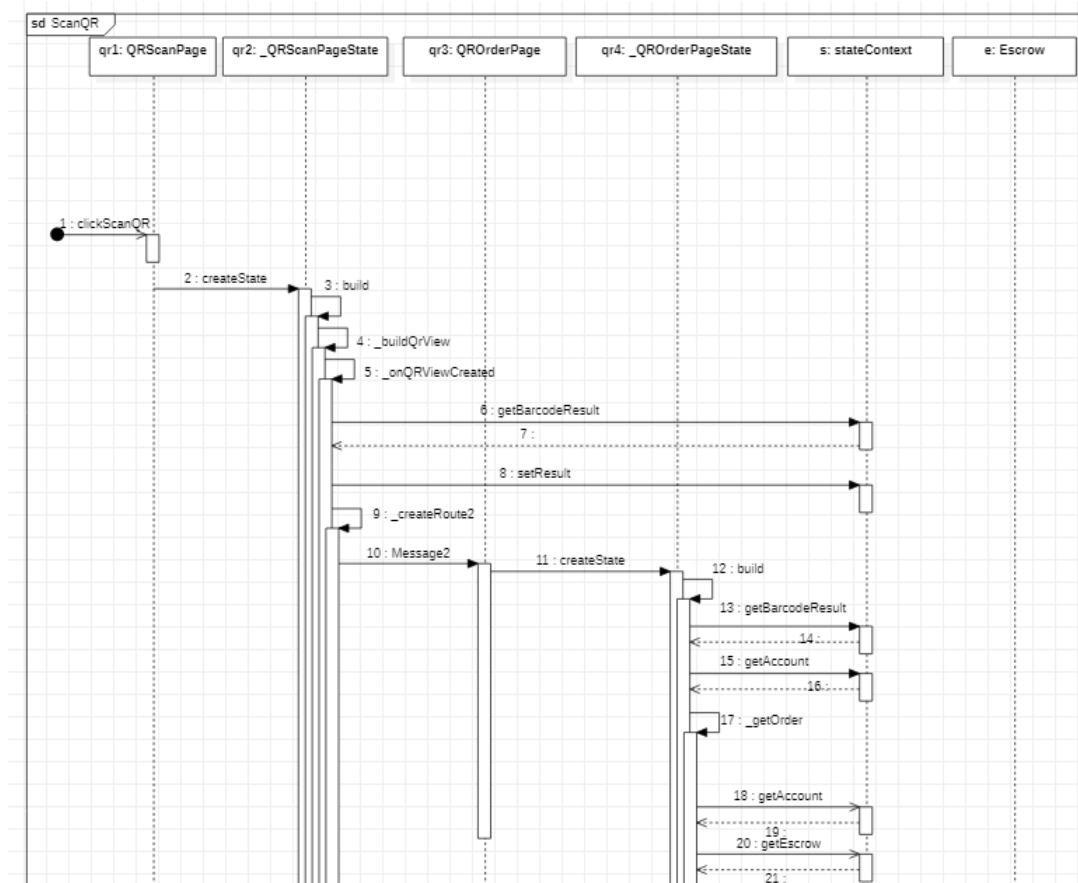
1. La DApp chiama la *_orderOperation(id, expr, orderAmount)* dall'oggetto StateContext(rappresentato nel codice da context) usando i parametri necessari per la Refund.
2. Viene ricevuto il risultato della funzione *askRefund(id)* e in base a ciò si aggiornano le informazioni(balance e stato ordine)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

3.3 Mobile

3.3.1 Diagramma delle classi

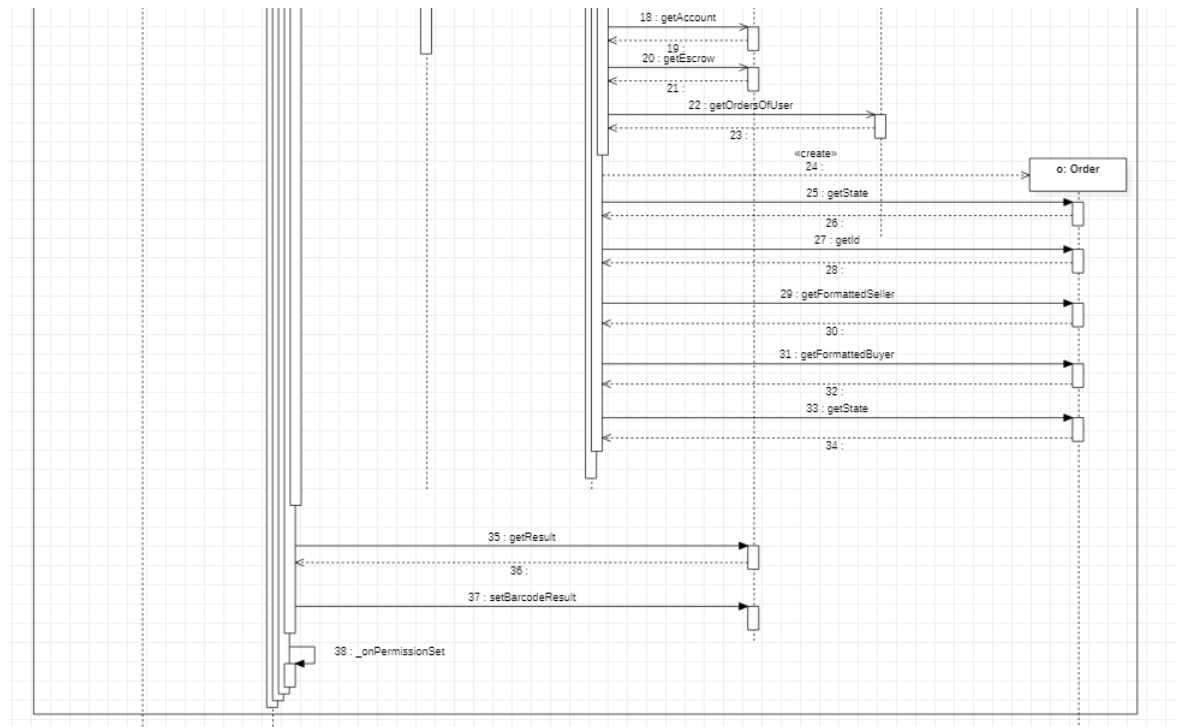
3.3.2 Diagrammi di sequenza

3.3.2.1 Scannerizzazione QRCode

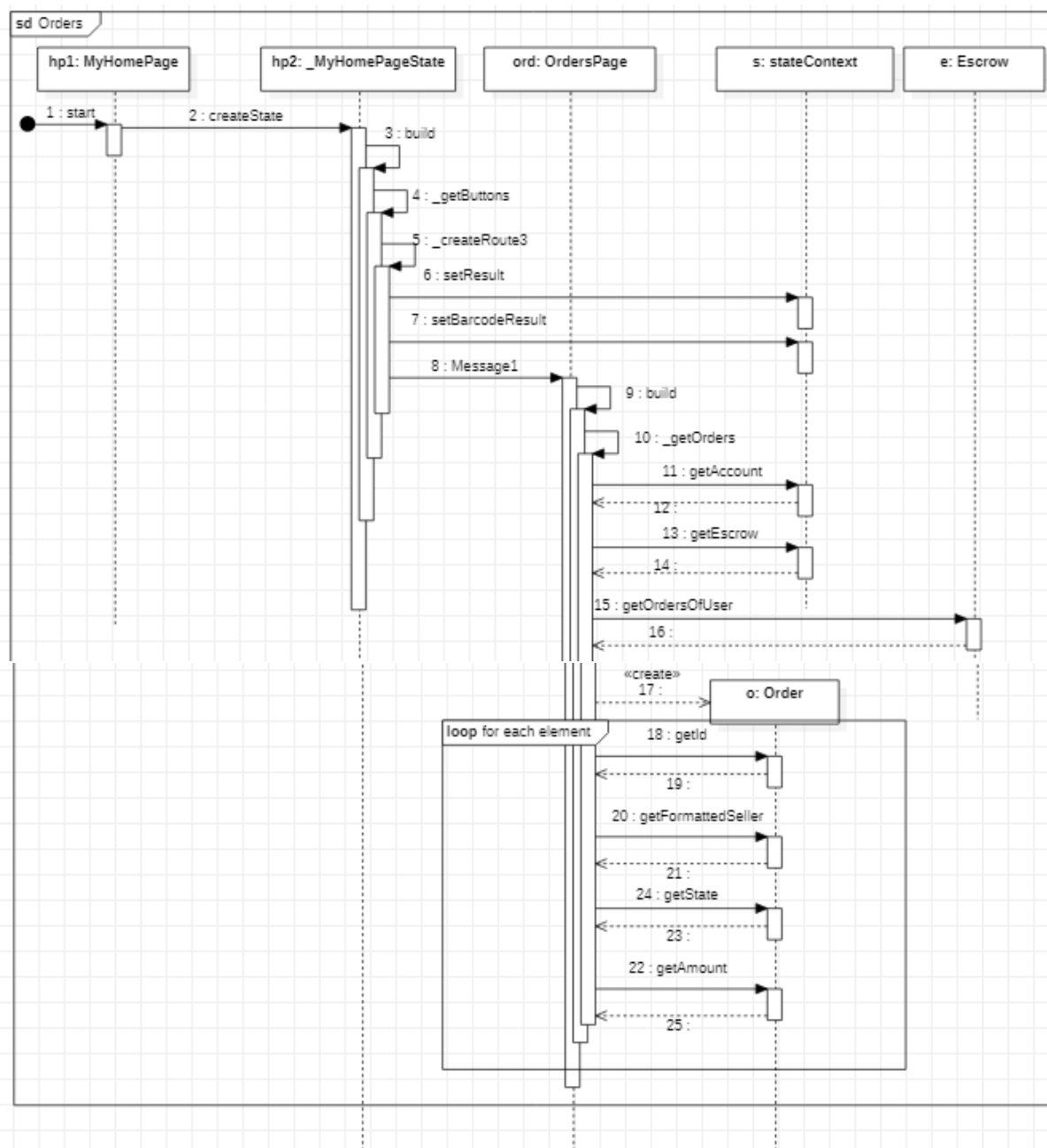


Il diagramma descrive i passi per la scannerizzazione del QRCode:

1. Viene cliccato il button per la scannerizzazione del QRCode, quindi viene creato lo stato del QRScanPage e i widget necessari.
2. DANILO SCIEMMO u.u
3. CONCORDO w.w
- 4.



3.3.2.2 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 la `_getOrders` che fa una chiamata allo `stateContext` per acquisire account e l'escrow su cui chiamare la `getOrdersOfUser` (metodo dello `SmartContract`).
3. Per ogni order vengono visualizzati i field `ID`, `address` del seller, stato e amount speso per l'ordine.

3.4 SmartContract

4 Punti possibili 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, ossia:

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

4.1 Fees rimborsate dall'e-commerce

Nello stato attuale, il sistema prevede che l'utente paghi le fees necessarie per l'uso di servizi aggiuntivi (ad es: 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 in campagne pubblicitarie dove può con ulteriori policy rendere il proprio e-commerce più competitivo.

Passi necessari:

- 1.
- 2.

4.2 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: milioni 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 reti come Ethereum e IPFS, in modo tale che sia sufficiente fare chiamate a questa libreria per ottenere i risultati già filtrati, spostando quindi la complessità dalle macchine degli utenti.

Passi necessari:

- 1.
- 2.

4.3 Sistema di reward

Si è pensato l'adozione di un sistema di reward affinché l'utente sia incentivato a pagare a rate utilizzando tokens durante fasi diversi 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 e il Seller può avere una visione più reattiva delle proprie entrate reali.

Passi necessari:

- 1.
- 2.

4.4 Verifica seller reali

La registrazione a Seller non richiede particolari requisiti, tuttavia questo crea una potenziale vulnerabilità, ossia lo 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) per poi verrà restituito al Seller dopo un determinato numero di ordine confermati.

L'efficacia di questo sistema è comprovato dalla sua adozione (in versioni diverse) da numerose piattaforme di successo come ad esempio PayPal.

Passi necessari:

- 1.
- 2.

4.5 Investimento fondi

Durante la creazione della transazione, i fondi vengono depositati e bloccati nello Smart-Contract, tuttavia in questo modo si sottrae un'opportunità sia alla rete che all'utente di poter guadagnare tramite investimenti perchè in questo lasso di tempo (circa il tempo di consegna del prodotto) in fondi sono effettivamente inutilizzabili.

Si vuole quindi dare la possibilità ai Buyers di poter scegliere le strategie di investimento per trarre vantaggio dei propri token anche quando sono bloccati.

Questa funzionalità è presente su applicazioni finanziarie come Yearn Finance.

Passi necessari:

- 1.
- 2.

5 Funzioni nella codifica

5.1 SmartContract

```
function createOrder(address payable seller) external payable

function confirmOrder(uint orderId) external
onlyBuyer orderExists(orderId) buyerIsOwner(orderId)

function deleteOrder(uint orderId) external
onlySeller orderExists(orderId) sellerIsOwner(orderId)

function askRefund(uint orderId) external
onlyBuyer orderExists(orderId) buyerIsOwner(orderId)

function refundBuyer(uint orderId) external
payable onlySeller orderExists(orderId) sellerIsOwner(orderId)

function registerAsSeller() external
```

5.2 WebApp

DAPP

```
_setListenerMetamaksAccount()
_setListenerNetworkChanged()
_initialize()
_changeNetwork()
_loadBlockchainData()
_refreshInfo(tx)
_initializeOrders()
_orderOperation(id, expr, orderAmount=0)
_getTotalOrders()
_getContractBalance()
_removeQRCode()
  Orders.jsx
applyFilters()
filterOrdersByAddress(address)
filterOrdersByState(ordersToFilter, state)
visualizeOrder(element)
RegisterSeller.jsx
_refreshInfo(tx)
_registerSeller()
StateContext.jsx
_connectWallet: async () => ,
```

```
_initialize: (userAddress) => ,  
_initializeEthers: async () => ,  
_setAddress: async () => ,  
_changeNetwork: async (networkName) => ,  
_wrongChain: () => ,  
_rightChain: () => ,  
_setListenerMetamaksAccount: () => ,  
_setListenerNetworkChanged: () => ,  
_connectWallet: async () =>  
_updateBalance: async () =>  
_isHisOrder: async (id) =>  
getOrderById: async (id) => ,  
_orderOperation: async (id, expr, orderAmount) => ,  
_getSellers: async () => ,  
_userIsSeller: async () => ,  
_getQRCode: async (order) => ;
```

5.3 LandingPage

```
createOrder(context, orderAmount, sellerAddress, afterConfirm)  
parseUrl()  
LandingPage()  
Notify( hasNotified )
```

5.4 Mobile

escrow.g.dart

```
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})

Ethereum credentials
Future<EthereumAddress> extractAddress()
Future<MsgSignature> signToSignature(UInt8List payload, int? chainId, bool isEIP1559 = false)
Future<String> sendTransaction(Transaction transaction)

MyHomepage
  _walletConnect()

OrdersPage
  Future<List<Order>> _getOrders()

QROrderPage
  Future<void> _confirmOrder(String orderId)
  Future<void> _askRefund(String orderId)
  Future<dynamic> _getOrder(int id)
  void makeRoutePage(BuildContext context, Widget pageRef)
  void _onQRViewCreated(QRViewController controller)
  void _onPermissionSet(BuildContext context, QRViewController ctrl, bool p)
```