
Università degli Studi di Catania
Corso di Laurea Magistrale in Informatica

---

Rosario Cannavò

# ValidiVault: ethereum blockchain-based property validation

---

Ingegneria dei Sistemi Distribuiti

---

Prof. E. A. Tramontana
Prof. A. F. Fornaia

---

**Anno Accademico 2023-2024**

# Indice


**1 Introduzione** **2**

**2 Design pattern e best practices** **3**
2.1 Microservizi e Docker . . . 3
2.2 Api Gateway . . . 4
2.3 Rate Limiter . . . 4
2.4 Circuit Breaker . . . 5
2.5 Role Based Access Control . . . 6
2.6 MetaMask Login Flow . . . 6
2.7 JWT Token Authentication . . . 8
2.8 Message Middleware . . . 9
2.9 Smart Contract: Proxy . . . 10
2.10 Smart Contract: Emergency Stop . . . 12

**3 Tecnologie utilizzate** **12**
3.1 Go Frameworks . . . 12
3.1.1 Gin . . . 13
3.1.2 Go Ethereum . . . 14
3.1.3 GoBreaker . . . 16
3.1.4 GoProxy . . . 18
3.2 Node.js Frameworks . . . 20
3.2.1 Express.js . . . 21
3.2.2 Web3.js . . . 21
3.3 Deployment . . . 22
3.3.1 MongoDB . . . 22
3.3.2 Redis . . . 22
3.3.3 RedisRateLimiter . . . 23
3.3.4 NATS message queue . . . 25
3.3.5 WebSocket . . . 26
3.4 Blockchain Development . . . 27
3.4.1 Solidity . . . 28
3.4.2 Truffle . . . 28
3.5 Blockchain Deployment . . . 29
3.5.1 Ganache . . . 29
3.5.2 Infura . . . 30

**4 Architettura** **31**
4.1 Struttura delle componenti . . . 33
4.1.1 Api Gateway . . . 33
4.1.2 RealServer . . . 37
4.1.3 Smart Contract . . . 38

4.1.4 Data Logger . . . . . 46

**5 User Workflow** **48**
5.1 Registrazione . . . . . 48
5.2 Login . . . . . 50
5.3 Admin Home page . . . . . 52
5.4 User Home page . . . . . 53
5.5 Stato della blockchain . . . . . 54
5.6 Logger . . . . . 55

**6 Conclusioni** **58**

# Abstract

ValidiVault è un ecosistema distribuito progettato per garantire l'integrità e la tracciabilità dei prodotti attraverso l'impiego della blockchain. Basato su microservizi e container Docker, l'architettura si concentra sull'autenticazione sicura degli utenti tramite varie modalità (Metamask, login/password) e sfrutta JSON Web Token (JWT) per controllare gli accessi. ValidiVault nasce dalla necessità di garantire un sistema sicuro e tracciabile per la gestione dei prodotti, rispondendo alle crescenti esigenze di autenticazione e convalida. L'utilizzo della blockchain e di tecnologie come Docker e microservizi mira a offrire un ecosistema flessibile, in grado di adattarsi a una vasta gamma di scenari applicativi, assicurando al contempo la sicurezza e l'integrità dei dati di prodotto.

# 1 Introduzione

Il presente rapporto dettaglia l'implementazione e lo sviluppo di "ValidiVault", un sistema distribuito fondato sull'architettura dei microservizi. L'intera infrastruttura è stata orchestrata tramite container Docker, offrendo flessibilità e gestione semplificata dell'ambiente di sviluppo. Il nucleo di questa soluzione è costituito da un API Gateway scritto in Go e sviluppato tramite il framework Gin. Questo gateway rappresenta il punto di accesso centrale per la gestione dell'autenticazione multipla degli utenti. L'autenticazione avviene attraverso il flow di autenticazione di Metamask, insieme alle credenziali di login e password. Le informazioni relative ai login degli utenti sono persistenti in un database MongoDB, mentre Redis agisce sia come servizio di caching per ottimizzare l'accesso a tali dati sia come "rate limiter" per garantire un controllo accurato degli accessi. L'API Gateway, una volta autenticati gli utenti, funge da proxy, implementando un sistema di controllo degli accessi basato sui ruoli (Role-Based Access Control). Tale controllo si attua mediante l'uso di JSON Web Token (JWT), che permette di distinguere tra utenti base e amministratori. Questo servizio comunica con un server reale implementato tramite Express.js, il quale interagisce con l'infrastruttura blockchain. La comunicazione con la blockchain avviene attraverso l'utilizzo di Web3.js e l'ABI (Application Binary Interface) di uno smart contract Solidity, deployato su una testnet sviluppata tramite Ganache. Questo consente agli amministratori di registrare i prodotti garantendone la validità, mentre agli altri utenti è consentito verificare la registrazione di un prodotto nella blockchain attraverso un ID univoco. Il processo di sviluppo e testing è stato eseguito utilizzando Truffle, fornendo un ambiente affidabile per il deploy dei contratti e la verifica delle funzionalità implementate. Inoltre,

per gestire in maniera efficiente le richieste, tutte le chiamate dirette all'API Gateway vengono instradate attraverso una coda NATS. Questa architettura permette l'indirizzamento verso un microservizio dedicato alla registrazione delle attività, agendo come logger che scrive su file le operazioni svolte, garantendo una tracciabilità accurata delle azioni nel sistema. L'insieme di microservizi e componenti descritto forma il sistema "ValidiVault", offrendo un ambiente distribuito, sicuro e ad alte prestazioni per la gestione e la convalida dei prodotti attraverso l'utilizzo della tecnologia blockchain.

# 2 Design pattern e best practices

Di seguito alcuni cenni sulle pratiche e i design pattern adottate per lo sviluppo del sistema.

## 2.1 Microservizi e Docker

I microservizi rappresentano una modalità di progettazione architetturale che suddivide un'applicazione complessa in componenti modulari e indipendenti chiamati servizi. Ogni microservizio esegue una specifica funzione o servizio dell'applicazione, operando in modo autonomo e comunicando con gli altri servizi tramite interfacce ben definite. Questo approccio favorisce la modularità, consentendo agli sviluppatori di lavorare su singoli servizi senza impattare l'intero sistema. L'utilizzo dei container è strettamente correlato all'approccio dei microservizi. Un container è un ambiente virtualizzato leggero e isolato che ospita un'applicazione e le sue dipendenze, garantendo la portabilità tra diversi sistemi operativi. Docker è una delle piattaforme più popolari per gestire i container, essa offre strumenti per la creazione, distribuzione e gestione di questi ambienti. L'integrazione di Docker con l'architettura a microservizi consente di impacchettare e distribuire facil-

mente ogni singolo microservizio in un container autonomo. Ciò consente una maggiore scalabilità e flessibilità nell'esecuzione e nella gestione dei servizi, oltre a semplificare il processo di deployment e l'isolamento delle singole funzionalità.

## 2.2 Api Gateway

Gli API Gateway agiscono come punto di accesso unificato per gestire e orchestrare le richieste provenienti dai client verso i servizi sottostanti all'interno di un'architettura a microservizi. Questi gateway fungono da intermediari, consentendo una gestione centralizzata delle richieste e delle risposte tra client e servizi backend. Uno dei principali compiti degli API Gateway è quello di semplificare e unificare l'accesso ai servizi sottostanti, esponendo un'interfaccia unificata e semplificata per i client. Essi permettono inoltre di gestire l'autenticazione, l'autorizzazione, il routing delle richieste e il monitoraggio delle performance dei servizi, offrendo un livello aggiuntivo di sicurezza e controllo. In un'architettura a microservizi, gli API Gateway possono anche eseguire funzioni di trasformazione dei dati, caching e orchestrazione delle richieste, contribuendo a ottimizzare le comunicazioni tra client e servizi. Questi componenti risultano fondamentali per garantire la scalabilità e l'affidabilità dei sistemi distribuiti, consentendo una gestione efficiente e centralizzata del traffico delle richieste all'interno dell'ecosistema dei microservizi.

## 2.3 Rate Limiter

I rate limiter sono strumenti utilizzati per controllare e limitare il numero di richieste o di operazioni che un utente può effettuare entro un determinato intervallo di tempo. All'interno di un'architettura distribuita come quella

basata su microservizi, i rate limiter sono essenziali per prevenire il sovraccarico dei servizi e proteggere il sistema da possibili attacchi di tipo DoS (Denial of Service) o altri tipi di abusi. Questi strumenti consentono di stabilire politiche che limitano la frequenza delle richieste da parte degli utenti, garantendo un uso equo delle risorse e evitando l'eccessivo utilizzo da parte di singoli client o applicazioni. L'implementazione di rate limiter può avvenire a diversi livelli dell'architettura, come ad esempio a livello di API Gateway, servizi o endpoint specifici. In pratica, i rate limiter monitorano il numero di richieste in ingresso da parte di un utente o di un client e, se superato un certo limite, possono rallentare, limitare o respingere ulteriori richieste per un certo periodo di tempo. Questo controllo svolge un ruolo fondamentale nella protezione dei servizi, garantendo un equilibrio nell'utilizzo delle risorse e prevenendo situazioni di sovraccarico che potrebbero compromettere le performance complessive del sistema.

## 2.4 Circuit Breaker

Il circuit breaker è un pattern progettato per gestire e migliorare la resilienza dei servizi all'interno di un'architettura distribuita. Simile al concetto di un interruttore automatico nelle reti elettriche, il circuit breaker monitora le chiamate ai servizi e interviene in caso di fallimento o di problemi prolungati. Quando un servizio o un'applicazione fallisce ripetutamente o presenta problemi, il circuit breaker si attiva, interrompendo temporaneamente la comunicazione con quel servizio in modo da prevenire ulteriori richieste che potrebbero causare danni o rallentamenti nel sistema. Questo meccanismo permette di gestire i tempi di degrado del servizio in maniera controllata e di recuperare la funzionalità in modo più rapido ed efficiente. Una volta attivato, il circuit breaker può adottare diverse strategie: può interrompere

temporaneamente le richieste, inoltrare risposte predefinite o tentare di utilizzare un percorso alternativo per soddisfare la richiesta dell'utente. Quando il servizio torna ad essere operativo, il circuit breaker può gradualmente ripristinare il flusso delle richieste in modo da valutare se il problema si è risolto in modo definitivo. In sostanza, il circuit breaker è uno strumento essenziale nell'architettura a microservizi poiché contribuisce a proteggere il sistema da potenziali guasti e a mantenere una migliore esperienza utente gestendo le situazioni di degrado dei servizi in maniera controllata.

## 2.5 Role Based Access Control

Il Role-Based Access Control (RBAC) è un modello di controllo degli accessi che si basa sulla definizione di ruoli e privilegi all'interno di un sistema. Questo modello organizza gli utenti in ruoli specifici all'interno dell'organizzazione e assegna loro determinati permessi o privilegi basati sul ruolo che ricoprono. L'approccio RBAC semplifica la gestione delle autorizzazioni, consentendo di definire in modo chiaro e gerarchico i diritti di accesso all'interno di un sistema. Ogni ruolo è associato a un insieme specifico di autorizzazioni o privilegi, e gli utenti acquisiscono automaticamente tali autorizzazioni una volta assegnati a un determinato ruolo. Ciò offre diversi vantaggi, tra cui una maggiore scalabilità, facilità di gestione e controllo degli accessi. Inoltre, il modello RBAC favorisce la sicurezza del sistema riducendo la complessità nella gestione degli accessi e garantendo un migliore controllo delle autorizzazioni a livello organizzativo.

## 2.6 MetaMask Login Flow

Il Metamask Login Flow rappresenta un metodo di autenticazione decentralizzato e sicuro che sfrutta il wallet Metamask, noto per consentire l'inte-

razione con le applicazioni basate su blockchain tramite il browser. Questo flusso di login consente agli utenti di autenticarsi in applicazioni decentralizzate (dApps) attraverso l'interfaccia Metamask. Quando un utente accede a una dApp che richiede l'autenticazione tramite Metamask, il flusso di login si attiva presentando una finestra popup che richiede all'utente di confermare l'accesso attraverso il suo wallet Metamask. Nel processo di autenticazione decentralizzata, l'applicazione genera una nonce univoca e la invia al client. Quest'ultimo propone la firma della nonce utilizzando il wallet Metamask e l'algoritmo ECDSA (Elliptic Curve Digital Signature Algorithm), che genera una firma crittografica unica combinando la nonce e la chiave privata del wallet. Successivamente, il client invia al server la firma della nonce insieme alla stessa nonce originale. Il server, utilizzando la chiave pubblica corrispondente al wallet Metamask associato all'account, verifica la firma della nonce con la nonce originale fornita. Se la verifica ha successo, conferma l'autenticazione dell'utente. Per garantire una maggiore sicurezza nel tempo, la nonce deve essere cambiata dopo ogni utilizzo, riducendo il rischio di possibili attacchi che potrebbero sfruttare la stessa firma precedentemente generata. Questo processo permette un'identificazione sicura dell'utente senza la necessità di trasmettere la chiave privata attraverso la rete, sfruttando la firma ECDSA reversibile e la verifica della nonce per stabilire l'autenticità dell'utente.

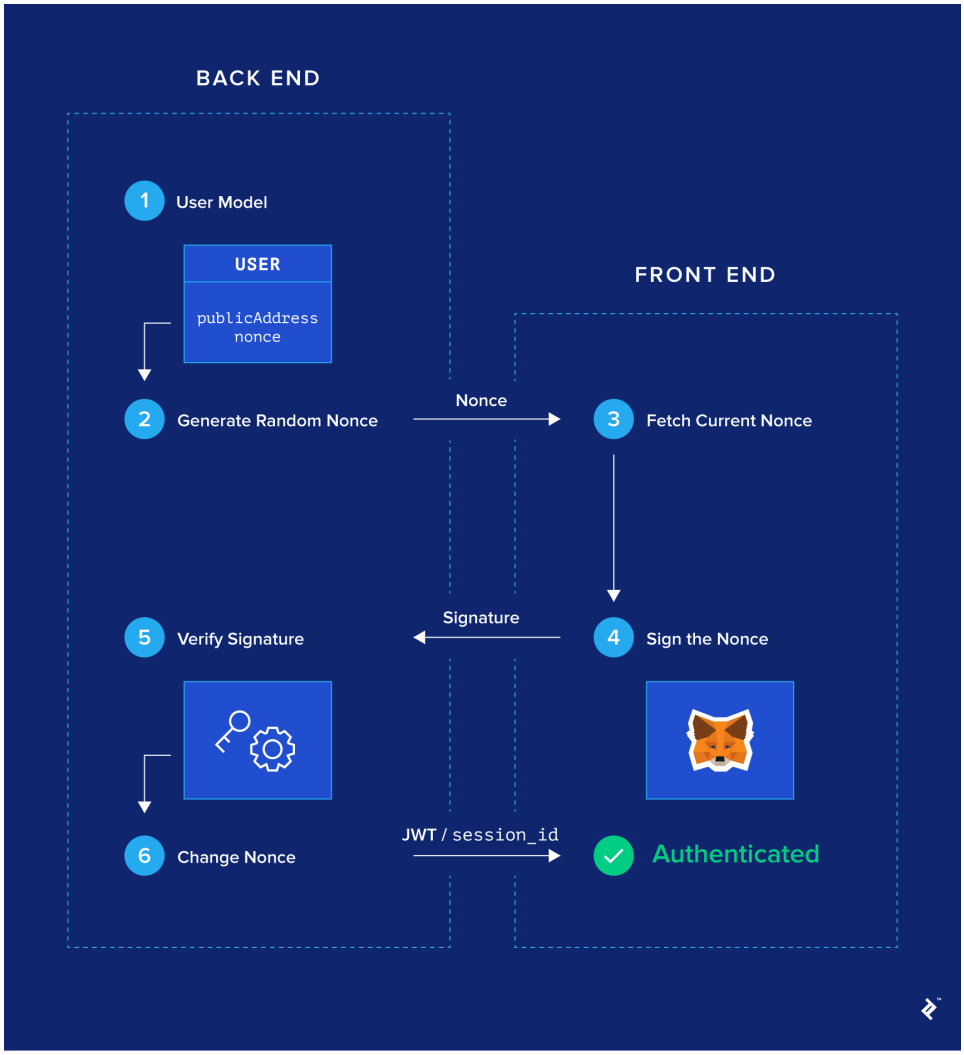


Figura 1: MetaMask Login Flow

## 2.7 JWT Token Authentication

JSON Web Token (JWT) è un formato aperto basato su JSON utilizzato per trasmettere informazioni tra due parti in modo sicuro e compatto. Nell'ambito dell'autenticazione, JWT è comunemente utilizzato per creare token di accesso che possono essere inviati tra client e server per verificare l'autenticità di un utente e concedere l'accesso a risorse protette. Un token JWT è

composto da tre parti separate da punti: l'intestazione (header), il payload e la firma. L'intestazione contiene il tipo di token e l'algoritmo di firma utilizzato. Il payload contiene le informazioni aggiuntive, come ad esempio l'identità dell'utente o le autorizzazioni. La firma è un valore crittografico generato utilizzando l'intestazione, il payload e una chiave segreta. Quando un utente effettua l'accesso al sistema, viene generato un token JWT contenente le informazioni necessarie per identificare l'utente. Questo token viene firmato e inviato al client. Ogni volta che il client richiede accesso a risorse protette, invia il token JWT al server attraverso l'header Authorization. Il server verifica la firma del token utilizzando la chiave segreta e decodifica il payload per ottenere le informazioni sull'utente e le autorizzazioni. L'utilizzo dei token JWT semplifica l'implementazione dell'autenticazione, poiché i token contengono tutte le informazioni necessarie per verificare l'autenticità dell'utente, riducendo la necessità di consultare il database per ogni richiesta. Inoltre, i token JWT sono autosufficienti e possono essere facilmente condivisi tra diverse parti del sistema, rendendo il processo di autenticazione più efficiente e scalabile.

## 2.8 Message Middleware

I middleware per i messaggi rappresentano un'infrastruttura fondamentale per facilitare la comunicazione affidabile tra diverse parti di un sistema distribuito. Questi strumenti agiscono come intermediari, consentendo alle applicazioni, ai servizi o ai dispositivi di scambiare informazioni in modo efficiente e affidabile. Uno dei principali vantaggi dei middleware per i messaggi è la capacità di gestire diversi modelli di comunicazione, come il modello pub/sub (pubblicazione/sottoscrizione) e il modello di coda. Questi modelli permettono ai mittenti di pubblicare messaggi su specifici canali o code,

mentre i destinatari interessati a tali messaggi possono sottoscriversi a quei canali o code per riceverli. Inoltre, i middleware per i messaggi forniscono soluzioni per problemi di scalabilità, affidabilità e interoperabilità. Possono garantire la distribuzione affidabile dei messaggi, la gestione della coda e dei flussi di lavoro, nonché facilitare la comunicazione tra applicazioni eterogenee, indipendentemente dal linguaggio di programmazione, dalla piattaforma o dalla posizione geografica. L'uso dei middleware per i messaggi consente una maggiore flessibilità nell'architettura di un sistema distribuito, migliorando la resilienza e la scalabilità, nonché semplificando la gestione delle comunicazioni asincrone tra le varie parti dell'applicazione o del sistema.

## 2.9 Smart Contract: Proxy

Il pattern Proxy in Solidity separa la logica di un contratto dalla sua gestione e dalla sua potenziale evoluzione. Questo è cruciale quando si desidera aggiornare la logica di un contratto senza modificare gli indirizzi con cui gli utenti interagiscono. Nel pattern del proxy, ci sono generalmente due contratti principali:

- **Contratto di Implementazione**: Contiene la logica del contratto, ovvero le funzioni e le operazioni che il contratto deve eseguire.
- **Contratto Proxy**: Agisce come intermediario tra gli utenti e il contratto di implementazione. Tutte le chiamate dei client vengono indirizzate a questo contratto.

Il contratto Proxy dispone di una funzione **fallback** che reindirizza tutte le chiamate non gestite a un contratto di implementazione specificato. Questo significa che se una chiamata non corrisponde a una funzione definita esplicitamente nel contratto Proxy, questa viene inoltrata al contratto di im-

plementazione. Quando si desidera aggiornare la logica del contratto, si crea una nuova versione del contratto di implementazione con le modifiche desiderate. Il contratto Proxy, tramite una funzione dedicata, aggiorna l'indirizzo del contratto di implementazione alla nuova versione. In sostanza, il fallback function in un contratto proxy agisce come un passaggio intermedio che reindirizza le chiamate non gestite al contratto di implementazione sottostante. Il vantaggio chiave di questo pattern è che gli utenti interagiscono costantemente con lo stesso contratto Proxy, nonostante la logica del contratto possa essere cambiata o aggiornata. Ciò significa che non è necessario modificare gli indirizzi con cui gli utenti si interfacciano, semplificando e mantenendo trasparenti le interazioni utente e migliorando la manutenibilità del sistema.

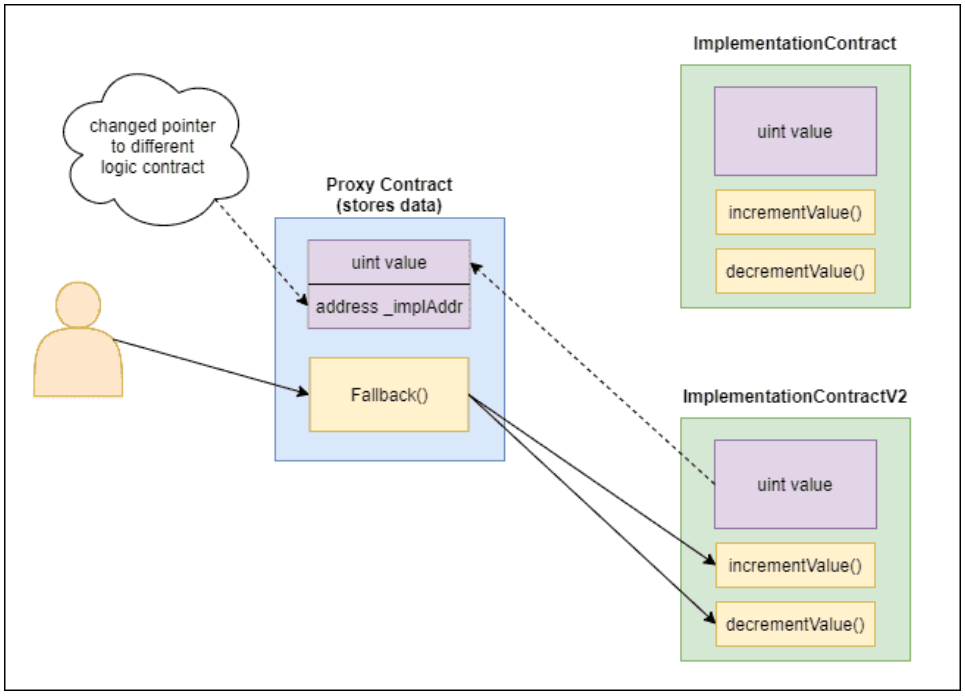


Figura 2: Solidity Proxy Design Pattern

### 2.10 Smart Contract: Emergency Stop

L'emergency stop in uno smart contract Solidity è un meccanismo progettato per interrompere temporaneamente le funzioni critiche o sensibili del contratto in situazioni di emergenza o di rischio significativo. Questo meccanismo è implementato per mitigare potenziali violazioni o pericoli imminenti. Solitamente, questo sistema si basa su un modificatore o una funzione all'interno del contratto che controlla se la pausa è stata attivata. Tale controllo viene eseguito all'inizio di ciascuna funzione vitale del contratto. Se la pausa è attiva, le funzionalità critiche vengono interrotte e non eseguite. L'abilitazione e la disabilitazione dell'emergenza stop vengono tipicamente gestite da un amministratore o da un ruolo autorizzato tramite una funzione dedicata. Questa funzione modifica uno stato interno del contratto per indicare se la pausa è attiva o disattivata. Durante la pausa d'emergenza, le funzioni critiche del contratto vengono sospese per mitigare potenziali danni o violazioni. Ad esempio, potrebbero essere disabilitate operazioni sensibili come trasferimenti di fondi o modifiche allo stato del contratto.

## 3 Tecnologie utilizzate

Di seguito vengono descritte le tecnologie utilizzate e la loro integrazione all'interno del sistema.

### 3.1 Go Frameworks

Per la realizzazione dell'API Gateway e del Logger è stato impiegato il linguaggio di programmazione Go (o Golang), un linguaggio efficiente e performante, noto per la sua velocità, la gestione delle concorrenze e la facilità di

sviluppo. Di seguito vengono riportati i framework utilizzati e viene illustrato il loro utilizzo all'interno del sistema.

### 3.1.1 Gin

Gin è un framework leggero per Go che facilita la creazione di API web efficienti. La struttura di base di Gin si basa su concetti chiave:

- **Router e Gruppi di Route**: Gin offre un sistema di routing robusto che consente di definire facilmente endpoint per gestire le richieste HTTP. I gruppi di route permettono di organizzare in modo gerarchico le route in base a percorsi comuni.
- **Middleware**: Gin supporta middleware, consentendo l'aggiunta di funzionalità comuni a livello di middleware. Questo può includere autenticazione, logging, gestione delle richieste, e molto altro ancora.
- **Contesti**: Gin utilizza i contesti per gestire i dati della richiesta e della risposta, permettendo la manipolazione dei dati in entrata e in uscita dalle route.
- **JSON e Serializzazione**: La libreria supporta la serializzazione e la deserializzazione dei dati in formato JSON, semplificando la trasmissione di dati tra il frontend e il backend.

Nel contesto dell'API Gateway, sono stati utilizzati i concetti di Gin route group per il Role-Based Access Control (RBAC) e di middleware per la gestione del Proxy e dell'autenticazione JWT.

- **Router group per RBAC**: I router group vengono impiegati specificamente per gestire le funzionalità di Role-Based Access Control

(RBAC). Questo implica la definizione di un sottoinsieme di route dedicate alla gestione dei controlli di accesso basati sui ruoli degli utenti o dei servizi all'interno dell'API Gateway. Le route all'interno del router group RBAC eseguono la logica necessaria per verificare i ruoli e i permessi degli utenti prima di autorizzare l'accesso a risorse specifiche.

- **Middleware**: Per altre funzionalità, come l'autenticazione basata su JWT e la gestione delle richieste Proxy , vengono utilizzati i middleware di Gin. Questi middleware vengono applicati globalmente o su un set più ampio di route. I middleware possono essere visti come dei pezzi di codice ripetuto che va posto davanti una o più route per effettuare determinate azioni.

### 3.1.2 Go Ethereum

La libreria Go Ethereum, comunemente nota come **Geth**, rappresenta uno strumento potente scritto in Go (Golang) utilizzato per interagire con la blockchain Ethereum. Essa offre una vasta gamma di funzionalità e strumenti che consentono agli sviluppatori e agli utenti di connettersi, interagire e sviluppare applicazioni sulla blockchain Ethereum.

- **Connettività alla Rete Ethereum**: Geth offre strumenti per connettersi alla rete Ethereum, permettendo ai nodi di sincronizzare la blockchain, partecipare alla rete e comunicare con altri nodi.

- **Gestione dei Portafogli e degli Account**: Geth permette agli utenti di creare, importare e gestire portafogli e account Ethereum. Ciò include la generazione di chiavi private, la gestione degli indirizzi e la firma delle transazioni.

- **Esecuzione di Smart Contract**: Geth supporta l'implementazione e l'esecuzione degli smart contract sulla blockchain Ethereum. Consente agli sviluppatori di caricare, distribuire e interagire con gli smart contract tramite transazioni.

- **Sviluppo e Testing di Applicazioni Blockchain**: È uno strumento fondamentale per gli sviluppatori che desiderano scrivere, testare e distribuire applicazioni decentralizzate (DApp) sulla rete Ethereum o su reti di test come Ropsten e Rinkeby.

- **Supporto alle API JSON-RPC**: Geth espone un'interfaccia JSON-RPC, permettendo alle applicazioni di interagire con il nodo Ethereum tramite richieste JSON. Questa API è utilizzata per eseguire operazioni come il recupero di informazioni sulla blockchain, la gestione delle transazioni, l'esecuzione degli smart contract, ecc.

Nell'API Gateway, la libreria Go Ethereum (Geth) è stata impiegata per diversi compiti cruciali:

- **Verifica delle Firme con Metamask**: Geth è stato utilizzato per la verifica delle firme digitali generate da Metamask. Questo processo consente di garantire l'autenticità delle transazioni effettuate dagli utenti tramite Metamask, verificando che le firme siano state generate correttamente da chiavi autorizzate.

- **Generazione della Nonce**: Geth è stato impiegato per generare e gestire le nonce, utilizzate come strumento di sicurezza per prevenire attacchi di ripetizione delle transazioni. Le nonce vengono generate in modo sicuro e utilizzate per garantire l'unicità delle transazioni nell'interazione con la blockchain di supporto.

- **Interazione con la Blockchain di Supporto**: Geth ha facilitato l'interazione con la blockchain di supporto, consentendo all'API Gateway di interrogare lo stato della blockchain.

### 3.1.3 GoBreaker

La libreria GoBreaker è un'implementazione del pattern Circuit Breaker, utilizzato per migliorare la resilienza e la stabilità delle applicazioni distribuite. Questo pattern viene impiegato per prevenire il fallimento a catena delle richieste e per gestire situazioni di errore prolungato o di sovraccarico di un servizio. Il funzionamento di GoBreaker, si basa su tre principali stati: Closed, Open e Half-Open. Questi stati determinano il comportamento del Circuit Breaker in base alle richieste in arrivo e alle risposte ottenute dai servizi. **Stato Chiuso (Closed):**

- Inizialmente, il Circuit Breaker è nello stato Closed, consentendo il normale flusso delle richieste al servizio.
- Durante questa fase, il Circuit Breaker monitora le richieste in uscita al servizio e valuta se le risposte ricevute superano una certa soglia di errore o di timeout.

**Stato Aperto (Open):**

- Se il Circuit Breaker rileva un numero eccessivo di errori o tempi di risposta troppo lunghi, passa nello stato Open.
- Quando è Open, il Circuit Breaker interrompe le richieste dirette al servizio, evitando così ulteriori chiamate che potrebbero aggravare la situazione.

**Stato Semi-Aperto (Half-Open):**

- Dopo un periodo di tempo predefinito, il Circuit Breaker può passare nello stato Half-Open.
- Durante lo stato Half-Open, il Circuit Breaker consente un numero limitato di richieste al servizio per testare la sua disponibilità.
- Se queste richieste hanno successo, il Circuit Breaker ritorna allo stato Closed, consentendo di nuovo il normale flusso di richieste.
- In caso di ulteriori fallimenti, il Circuit Breaker torna allo stato Open, evitando ulteriori chiamate al servizio per un periodo ulteriore.

Il passaggio di stato è rappresentato dal seguente schema:

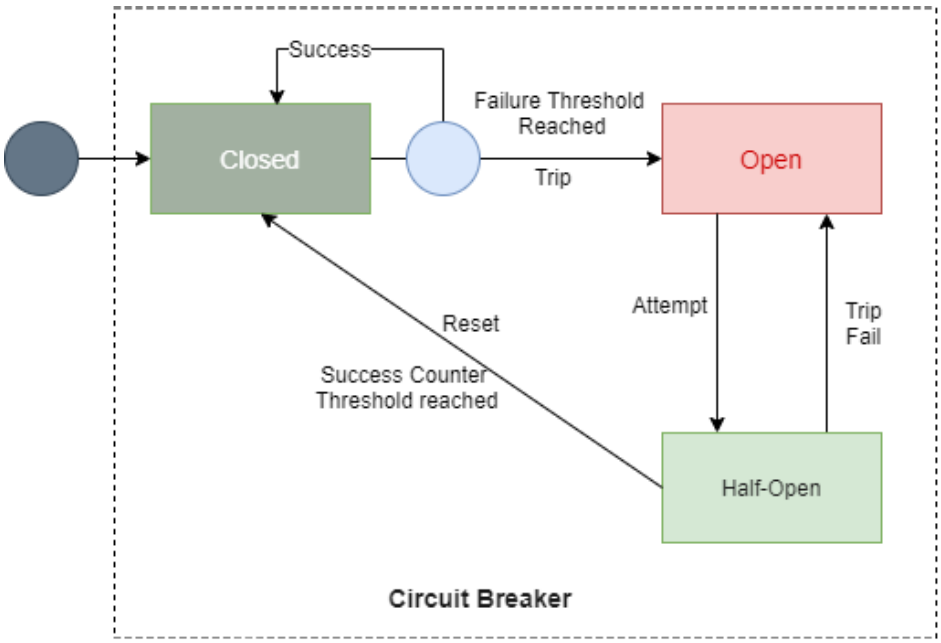


Figura 3: Circuit Breaker state change

Nell'API Gateway, GoBreaker è stato utilizzato come filtro primario per gestire tutte le richieste dirette al server reale attraverso il Reverse Proxy.

Questa integrazione del Circuit Breaker è stata posta come strato di protezione per controllare e regolare il flusso delle richieste in arrivo verso il server reale. Il funzionamento del GoBreaker nell'API Gateway, posizionato prima delle richieste indirizzate al server reale attraverso il proxy, segue lo stesso principio di funzionamento descritto in precedenza. Il Circuit Breaker monitora la salute del servizio di destinazione, valutando le risposte delle richieste e attivando lo stato Open se rileva un numero eccessivo di errori o un ritardo nelle risposte al di sopra di una soglia prestabilita. Questo approccio consente di proteggere il server reale da possibili sovraccarichi o malfunzionamenti, limitando il flusso delle richieste in caso di problemi rilevati dal Circuit Breaker. Il GoBreaker svolge quindi un ruolo critico nell'assicurare la stabilità e la resistenza dell'API Gateway, garantendo che il server reale non venga sovraccaricato da un elevato numero di richieste in caso di problemi rilevati durante l'elaborazione delle richieste. Essendo le richieste effettuate tramite il circuit breaker dirette ad un Proxy, è stato inoltre necessario realizzare un meccanismo per la ricezione delle risposte ai client originali da parte del real server, dando modo al circuit breaker di monitorare la situazione della rete senza impattare le prestazioni.

### 3.1.4 GoProxy

Il proxy in Go, implementato con la libreria HTTP standard, funziona come intermediario tra un client e un server di destinazione. In particolare, nell'API Gateway, è stato utilizzato per instradare tutte le richieste al server reale dopo l'autenticazione. Ecco un approfondimento:

**Gestione delle Richieste dell'API Gateway:**

- **Accettazione delle Richieste del Client:**
  - Il proxy crea un server HTTP in ascolto su una determinata porta.

  - Quando il client invia una richiesta HTTP all'API Gateway, il server proxy la riceve.

- **Analisi e Manipolazione delle Richieste:**
  - Il proxy esamina la richiesta in ingresso, inclusi URL, metodi HTTP (GET, POST, ecc.), header e dati.

- **Inoltraggio delle Richieste al Server Reale:**
  - Dopo l'autenticazione tramite l'API Gateway, il proxy instrada le richieste autenticate al server reale.
  - Utilizzando un client HTTP, il proxy crea e inoltra nuove richieste HTTP verso il server reale.

- **Gestione della Risposta dal Server Reale:**
  - Attende la risposta dal server reale, la riceve e ne analizza i dati e gli header.

- **Inoltro della Risposta al Client Originale:**
  - Una volta ricevuta la risposta dal server reale, il proxy la inoltra al client originale che ha effettuato la richiesta all'API Gateway.

**Ruolo nell'API Gateway:** Il proxy, all'interno dell'API Gateway, agisce come un intermediario trasparente che gestisce tutte le richieste autenticate, dirigendole al server reale per l'elaborazione. Questa configurazione consente di controllare e ottimizzare il flusso delle richieste, fornendo un'interfaccia sicura tra il client e il server, mentre il proxy gestisce la trasmissione sicura e il corretto instradamento delle richieste autenticate. Il codice che implementa il Proxy, protetto dal circuit breaker è il seguente:

```go
func createReverseProxy(remote *url.URL, headers http.Header, proxyPath string) *httputil.ReverseProxy
{
    proxy := httputil.NewSingleHostReverseProxy(remote)

    proxy.Director = func(req *http.Request) {
        req.Header = headers
        req.Host = remote.Host
        req.URL.Scheme = remote.Scheme
        req.URL.Host = remote.Host
        req.URL.Path = proxyPath
    }

    return proxy
}

func handleResponse(proxy *httputil.ReverseProxy, w http.ResponseWriter, r *http.Request) int {
    rrw := models.NewResponseRecorderWriter(w)
    proxy.ServeHTTP(rrw, r)
    capturedStatus := rrw.StatusCode

    message := fmt.Sprintf("Timestamp: %s | Handler: %s | Status: %d | Response: %s",
time.Now().UTC().Format(time.RFC3339), "proxy_handler/handleResponse", capturedStatus, rrw.Body)
    nats.NatsConnection.PublishMessage(message)

    return capturedStatus
}

func ProxyHandler(c *gin.Context) {

    remote, err := url.Parse(config.ProxyDestination)

    if err ≠ nil {
        panic(err)
    }

    proxy := createReverseProxy(remote, c.Request.Header, c.Param("proxyPath"))
    _, errcb := circuit_breaker.CircuitBreaker.Execute(func() (interface{}, error) {

        status := handleResponse(proxy, c.Writer, c.Request)

        if status < 200 || status ≥ 300 {
            return nil, errors.New("server error")
        }

        return nil, nil

    })

    if errcb ≠ nil {
        fmt.Println("circuit breaker error", errcb)
    }

}
```

Figura 4: Codice Proxy e Circuit Breaker

## 3.2 Node.js Frameworks

Per la realizzazione del servizio principale, è stato scelto di utilizzare Node.js in quanto richiede un'integrazione stretta con lo smart contract. La decisione è stata presa considerando che la maggior parte delle librerie per l'interazione

con gli smart contract sono sviluppate in JavaScript. L'utilizzo di Node.js agevola l'interazione diretta con gli smart contract implementati in Solidity, permettendo una più agevole integrazione e sfruttando le librerie esistenti per la gestione delle transazioni e la comunicazione con la blockchain.

### 3.2.1 Express.js

Express.js è stato adottato nel progetto per esporre due endpoint (root) che rappresentano le interazioni disponibili per gli utenti con lo smart contract. Utilizzando Express.js, è stato possibile creare in modo rapido e efficiente un'interfaccia web che gestisce le richieste provenienti dal Proxy e le traduce in operazioni dirette con lo smart contract. Gli endpoint forniti consentono agli utenti di interagire in modo sicuro e strutturato con le funzionalità offerte dallo smart contract sulla blockchain.

### 3.2.2 Web3.js

Web3.js è una libreria JavaScript fondamentale per l'interazione con le blockchain Ethereum, consentendo l'accesso e l'esecuzione di operazioni all'interno dello smart contract tramite l'ABI (Application Binary Interface). L'ABI funge da specifica per la comunicazione con il contratto, definendo i metodi, i parametri e i tipi di dati accettati e restituiti. Nel contesto dello sviluppo, Web3.js è stato adottato come componente centrale delle API esposte tramite Express.js. Queste API consentono agli utenti di accedere e manipolare il contratto, consentendo azioni come la registrazione di prodotti o la verifica dello stato dei prodotti sulla blockchain Ethereum. Utilizzando l'ABI, Web3.js agevola l'esecuzione di queste operazioni, facilitando la trasmissione dei dati tra il backend e il contratto stesso. È attraverso Web3.js che il server comunica con il contratto, garantendo l'accuratezza e la coerenza delle opera-

zioni eseguite, nonché la sincronizzazione dei dati sulla blockchain. In questo modo, Web3.js rappresenta il fulcro delle interazioni tra il server e il contratto, assicurando che le operazioni definite vengano eseguite correttamente sulla blockchain Ethereum.

## 3.3 Deployment

### 3.3.1 MongoDB

MongoDB è un database NoSQL flessibile e scalabile che utilizza il modello di dati basato su documenti. Nel contesto dello sviluppo, è stato scelto come sistema di archiviazione per memorizzare le credenziali degli utenti. Essendo un database flessibile, MongoDB permette di memorizzare dati in documenti JSON, il che lo rende ideale per salvare informazioni strutturate in un formato flessibile. Nello specifico, è stato utilizzato per conservare le credenziali degli utenti in modo sicuro e scalabile. L'approccio flessibile di MongoDB consente di gestire le credenziali degli utenti in documenti separati, garantendo allo stesso tempo una rapida indicizzazione e una facile scalabilità, essenziale soprattutto in contesti in cui è richiesta una gestione sicura delle credenziali degli utenti. Utilizzare MongoDB per questa finalità offre un'opportunità di gestire in modo efficiente e affidabile le informazioni di autenticazione degli utenti nel sistema.

### 3.3.2 Redis

Redis è un sistema di archiviazione chiave-valore in memoria, noto per la sua estrema velocità e versatilità. Nello sviluppo, è stato adottato come strumento di caching per il database MongoDB. L'utilizzo di Redis come cache per MongoDB permette di migliorare le prestazioni complessive del sistema. Conservando temporaneamente i dati frequentemente accessati in

memoria, Redis riduce il tempo di accesso ai dati rispetto al caricamento da un database su disco, garantendo un accesso più rapido alle informazioni. Questo approccio di caching aiuta a ottimizzare le prestazioni del sistema, specialmente quando si tratta di dati che vengono letti o scritti frequentemente. Redis, con la sua struttura di archiviazione in memoria ad alte prestazioni, offre un modo efficiente per memorizzare temporaneamente dati critici e ridurre il carico sul database principale, migliorando così la reattività complessiva dell'applicazione.

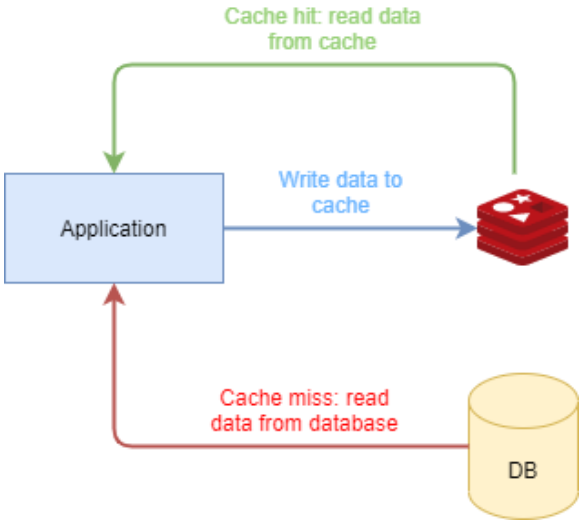


Figura 5: Redis Cache

### 3.3.3 RedisRateLimiter

Il Redis rate limiter è uno strumento utilizzato per limitare il numero di richieste che possono essere effettuate verso un'applicazione o un endpoint in un determinato intervallo di tempo. Nel contesto del servizio realizzato, è stato impiegato come un meccanismo di difesa all'interno dell'API Gateway per proteggere da possibili attacchi di tipo DDoS o da un eccessivo carico di richieste. L'API Gateway, grazie all'uso del Redis rate limiter, è stato

configurato per accettare solo un numero specifico di richieste entro un determinato intervallo di tempo da parte di un singolo utente o da un IP. Se il numero massimo di richieste viene superato, il rate limiter blocca temporaneamente ulteriori richieste provenienti dalla stessa fonte, garantendo così che l'API non venga sovraccaricata o soggetta a un utilizzo eccessivo. Questo meccanismo di limitazione del tasso di richieste, implementato utilizzando Redis come memoria di archiviazione dei dati di controllo, contribuisce a proteggere l'API Gateway, mantenendo una gestione equilibrata e controllata del traffico in ingresso. Questo, a sua volta, aiuta a mantenere le prestazioni dell'API e a garantire un funzionamento stabile e affidabile del sistema nel suo complesso.

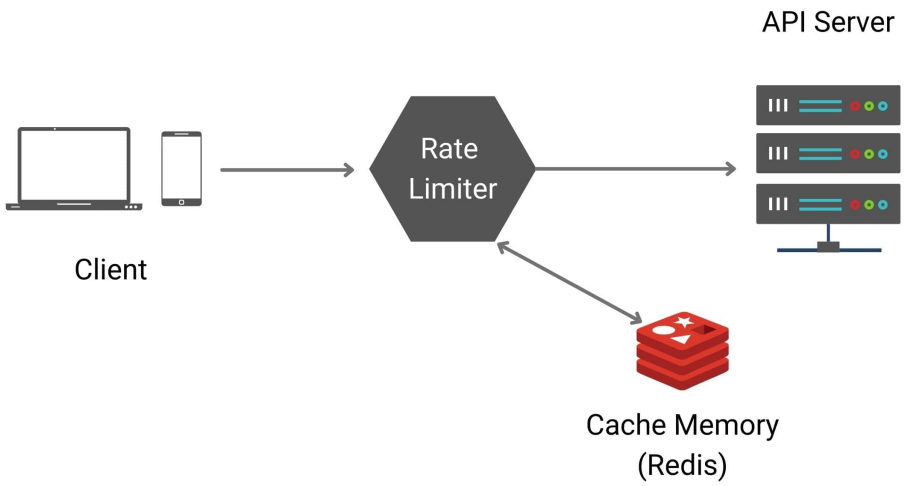


Figura 6: Redis Rate Limiter

Il codice del middleware che si occupa di implementare il rate limiter è il seguente:

```
type RateLimitMiddleware struct {
    RedisLimiter *ratelimiter.RedisRateLimiter
}

func NewRateLimitMiddleware() *RateLimitMiddleware {
    return &RateLimitMiddleware{ratelimiter.SetupRedisRateLimiter()}
}

const RateRequest = "rate_request_%s"

func (r *RateLimitMiddleware) Handler() gin.HandlerFunc {
    return func(c *gin.Context) {
        res, err := r.RedisLimiter.Allow(c, fmt.Sprintf(RateRequest, "userName"), redis_rate.Limit{
            Rate:   1, //max req per client
            Burst:  5,
            Period: time.Second,
        })

        if err != nil || res.Allowed <= 0 {

            message := fmt.Sprintf("Timestamp: %s | Handler: %s | Status: %d | Response: %s",
time.Now().UTC().Format(time.RFC3339), "middleware/RateLimiter", http.StatusTooManyRequests, "error:
Too many requests")
            nats.NatsConnection.PublishMessage(message)

            c.AbortWithStatusJSON(http.StatusTooManyRequests, gin.H{"error": "Too many requests"})
            return
        }

        c.Next()
    }
}
```

Figura 7: Codice Redis Rate Limiter

### 3.3.4 NATS message queue

NATS è un sistema di messaggistica leggero e ad alte prestazioni che utilizza un protocollo leggero e efficiente chiamato **NATS Protocol**. È progettato per la messaggistica e la comunicazione distribuita ad alte prestazioni. NATS viene utilizzato nell'architettura per gestire la registrazione dei codici di risposta delle richieste effettuate alle root dell'API Gateway. Nello specifico, NATS è stato configurato per agire come una coda di messaggi in grado di ricevere e instradare i codici di risposta delle richieste effettuate alle varie root dell'API Gateway. Ogni volta che una richiesta viene eseguita con successo o fallisce, il codice di risposta corrispondente viene correlato di un timestamp e altri dati satellite utili al logging e inviato alla coda di messaggi

NATS. Una volta ricevuti i codici di risposta, NATS si occupa di inoltrarli a un servizio di logging, il quale registra e archivia queste informazioni su un file di log. Questa architettura a coda di messaggi consente una gestione efficiente e asincrona dei codici di risposta, garantendo che vengano registrati senza influire sul flusso principale dell'applicazione. L'utilizzo di NATS per la gestione dei messaggi di risposta permette un'ottima scalabilità e una gestione affidabile dei log delle richieste, consentendo al sistema di mantenere un tracciamento dettagliato delle interazioni con l'API Gateway senza compromettere le prestazioni complessive.

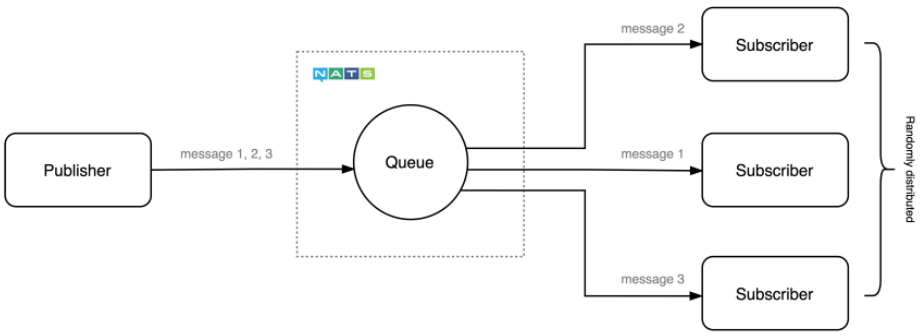


Figura 8: NATS message queue flow

### 3.3.5 WebSocket

I WebSockets rappresentano un protocollo di comunicazione avanzato che facilita lo scambio di dati bidirezionale e in tempo reale tra un client e un server. A differenza delle connessioni HTTP tradizionali, i WebSockets consentono una comunicazione persistente su un singolo canale, eliminando la necessità di creare continuamente nuove connessioni. Questo metodo ottimizza l'efficienza della comunicazione, permettendo sia al client che al server di inviare messaggi in maniera istantanea senza l'onere degli overhead tipici delle richieste HTTP standard. Grazie a questa caratteristica, i WebSoc-

kets sono ampiamente utilizzati in applicazioni che richiedono una risposta in tempo reale, come chat in tempo reale, giochi online, aggiornamenti live di dati finanziari e piattaforme di collaborazione. La loro capacità di mantenere una connessione persistente e di facilitare una comunicazione fluida e bidirezionale li rende una tecnologia fondamentale per molte applicazioni web moderne. Il protocollo WebSocket è riassunto nel seguente schema:

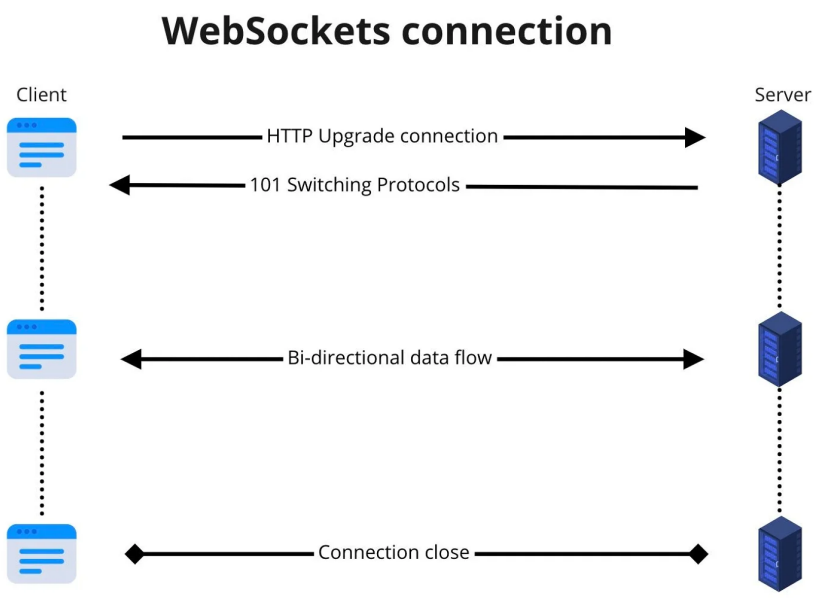


Figura 9: WebSocket flow

## 3.4 Blockchain Development

Questa sezione è dedicata alla presentazione degli strumenti impiegati durante lo sviluppo e il testing degli smart contract.

### 3.4.1 Solidity

Solidity è un linguaggio di programmazione orientato ai contratti intelligenti progettato specificamente per Ethereum e altre piattaforme blockchain compatibili con Ethereum Virtual Machine (EVM). Si basa su una sintassi simile a JavaScript e viene utilizzato per scrivere contratti intelligenti che definiscono le regole e le logiche di funzionamento all'interno di una blockchain. Solidity permette di definire le operazioni e le proprietà dei contratti, consentendo la creazione di applicazioni decentralizzate (DApps) e l'esecuzione di codice autonomo all'interno dell'ambiente blockchain.

### 3.4.2 Truffle

Truffle è un framework di sviluppo per Ethereum che semplifica la creazione, il testing e il deploy degli smart contract. È stato impiegato nel progetto per gestire il ciclo di vita completo degli smart contract, dalla scrittura del codice alla sua distribuzione e al testing. Truffle fornisce un ambiente di sviluppo integrato (IDE) che permette agli sviluppatori di scrivere, compilare, testare e distribuire gli smart contract in modo efficiente e strutturato. Grazie alle sue funzionalità avanzate e agli strumenti di testing integrati, Truffle ha agevolato il processo di sviluppo e testing degli smart contract. Il framework offre un insieme di strumenti che semplificano il processo di deploy, consentendo agli sviluppatori di scrivere, compilare e distribuire gli smart contract su una rete Ethereum o una blockchain di test. Inoltre, Truffle integra strumenti di testing come Mocha e Chai per la scrittura e l'esecuzione di test automatizzati degli smart contract. Questi strumenti consentono agli sviluppatori di definire e eseguire test per verificare la correttezza e il comportamento degli smart contract in diversi scenari e condizioni. L'uso di Truffle, Mocha e Chai insieme ha permesso di automatizzare il processo

di testing degli smart contract, garantendo che le funzionalità implementate siano conformi alle aspettative e alle specifiche stabilite durante lo sviluppo.

## 3.5 Blockchain Deployment

### 3.5.1 Ganache

Ganache è un ambiente di sviluppo personale blockchain che offre una blockchain Ethereum locale, adatta per scopi di sviluppo e testing. Nello sviluppo, Ganache è stato utilizzato insieme a MetaMask per fornire una blockchain locale su cui sviluppare e testare gli smart contract. La correlazione tra Ganache e MetaMask è stata fondamentale: Ganache ha fornito una blockchain locale che simulava l'ambiente Ethereum, mentre MetaMask, tramite una rete personalizzata, consentiva agli sviluppatori di interagire con questa blockchain locale attraverso il browser. Truffle, a sua volta, è stato utilizzato per gestire il deploy degli smart contract sulla blockchain simulata da Ganache. Grazie a questa combinazione di strumenti, è stato possibile scrivere, testare e distribuire gli smart contract in un ambiente controllato e simulato, consentendo lo sviluppo e la verifica delle funzionalità senza l'uso di una rete Ethereum pubblica o di test.

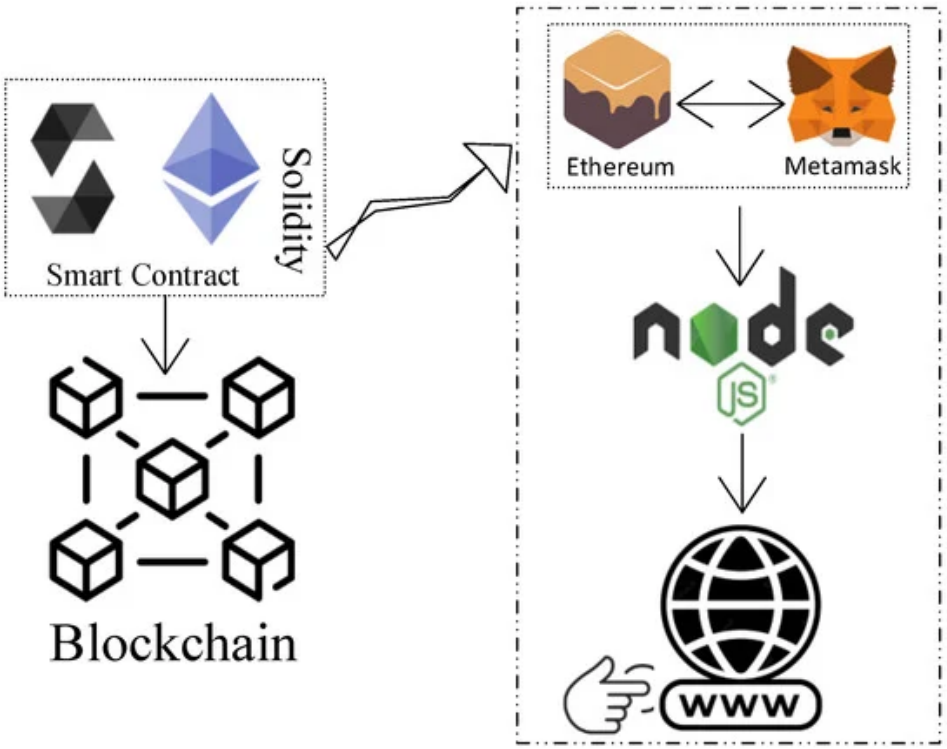


Figura 10: Ganache and Metamask Integration

### 3.5.2 Infura

Infura è un servizio di infrastruttura che fornisce accesso alle reti blockchain, incluso Ethereum, senza la necessità di eseguire un nodo completo in locale. Nel contesto del servizio realizzato, Infura è stato utilizzato per effettuare testing sulla rete Sepolia. La rete Sepolia è una testnet Ethereum che consente agli sviluppatori di testare e sviluppare smart contract senza utilizzare risorse o valuta reale sulla rete principale Ethereum. Infura ha fornito accesso a questa testnet, dando la possibilità di interagire con essa e di eseguire test, inclusi test di deploy e di funzionalità degli smart contract, senza dover configurare e gestire un nodo Ethereum locale. Per effettuare il deploy del contratto su una testnet Sepolia, dunque senza dover operare in locale, il pri-

mo passo necessario è quello di ottenere un'API KEY per il progetto previa registrazione al servizio infura. Dopo aver effettuato la registrazione si otterrà un'API key e sarà possibile interagire con la testnet Sepolia attraverso il seguente URL:

$$https://sepolia.infura.io/v3/\${API\_KEY}$$

A questo punto effettuare il deploy dello smart contract può essere effettuato banalmente impostando l'account sender e il link ottenuto nel file truffle-config.js.

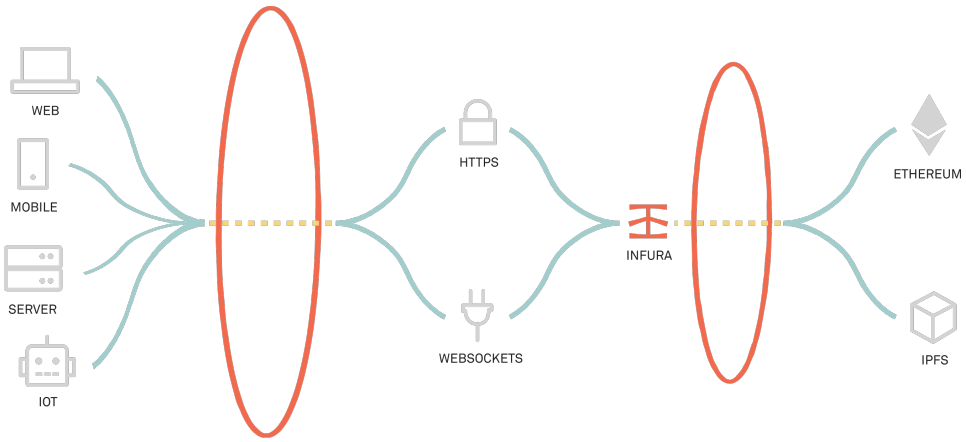


Figura 11: Infura

# 4 Architettura

In questa sezione verrà descritta nel dettaglio l'architettura del sistema e come avvengono le interazioni fra le componenti. Ogni componente è da intendersi come un microservizio a se stante che viene eseguito all'interno di un container Docker isolato. La comunicazione avviene tramite una overlay network gestita dal Docker Deamon, solo i container che eseguono i microser-

vizi fanno parte di questa rete e possono dunque interagire tra di loro, questo garantisce isolamento dai servizi esterni. La struttura dell'architettura e le relazioni delle azioni effettuate sono rappresentate dal seguente diagramma:

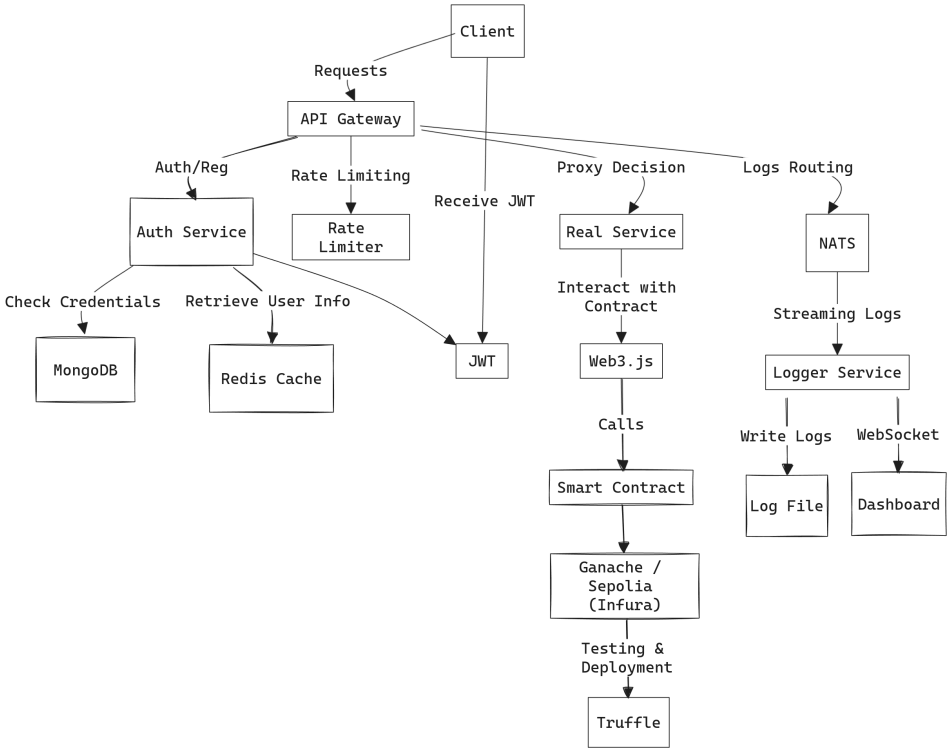


Figura 12: Architettura del sistema

Di seguito viene riportata la struttura interna di ogni componente.

## 4.1 Struttura delle componenti

### 4.1.1 Api Gateway

L'Api Gateway espone diverse root protette da diversi livelli di sicurezza. Di seguito l'elenco completo e le funzionalità. Si noti che tutte le root dell'api gateway sono protette da un **rate limiter** applicato a livello di reverse Proxy che permette al più un numero configurabile di richieste da parte di un client in modo da evitare attacchi o sovraccarichi inutili del sistema. Inoltre ogni risposta alle richieste dell'api gateway viene inoltrata su NATS per effettuare il logging. Ogni volta che avviene una richiesta successiva a quella di registrazione o accesso viene sempre invocato il middleware che richiede il token JWT, in questo modo un utente sarà sempre autenticato, inoltre, sarà possibile recuperare dal token anche il **role** associato in modo da filtrare sempre i permessi che ha a disposizione l'utente che sta interagendo con il sistema.

- /: questa root espone la pagina html di login e permette di effettuare l'accesso al sistema previa registrazione tramite le credenziali e l'interazione con il plugin metamask del browser.

- /**signup**: questa root espone la pagina di registrazione al sistema e permette di inserire le proprie credenziali per la registrazione al sistema.

- /**registration**: questa root riceve in input le informazioni del form di registrazione e l'indirizzo dell'account associato alla sessione metamask dell'utente, quando invocata vengono validate le informazioni passate e in caso di esito affermativo verrà generata una nonce random utile all'autenticazione tramite MetaMask e le informazioni dell'uten-

te verranno registrate nel database mongoDB. Durante il processo di registrazione, inoltre, viene appreso e impostato il ruolo dell'utente che vuole effettuare il login effettuando un controllo con l'indirizzo MetaMask usato dall'utente e registrato nel database.

- /**login**: prende in input le credenziali inserite nel form di login ed effettua una verifica tramite la cache Redis e in caso di miss tramite il database, in caso di esito affermativo scatena il workflow di autenticazione MetaMask recuperando la nonce dal database per ritornarla al client. Alla fine se anche il processo MetaMask va a buon fine, verrà restituito al client un **Token JWT** che gli sarà utile per navigare all'interno del sistema rimanendo autenticato. Si noti che durante la fase di login viene recuperato il ruolo dell'utente, che viene vincolato all'interno del token JWT in modo da essere recuperabile da ogni richiesta autenticata che viene effettuata, vengono inoltre inseriti il nome utente e l'indirizzo fornito. In questo modo il token rappresenta un'unica informazione protetta (cifrata) che permette di avere un quadro completo sull'utente che sta interagendo con il sistema, evitando interrogazioni multiple al database da parte delle root successive.

- /**verify-signature**: si occupa di verificare la firma della nonce effettuta dal client tramite il plugin MetaMask, in caso di esito affermativo genera e memorizza una nuova nonce per l'utente e ritorna un token JWT valido al client, il token incorporerà tra le varie informazioni anche il ruolo dell'utente in modo da facilitare tramite un middleware il role based access control.

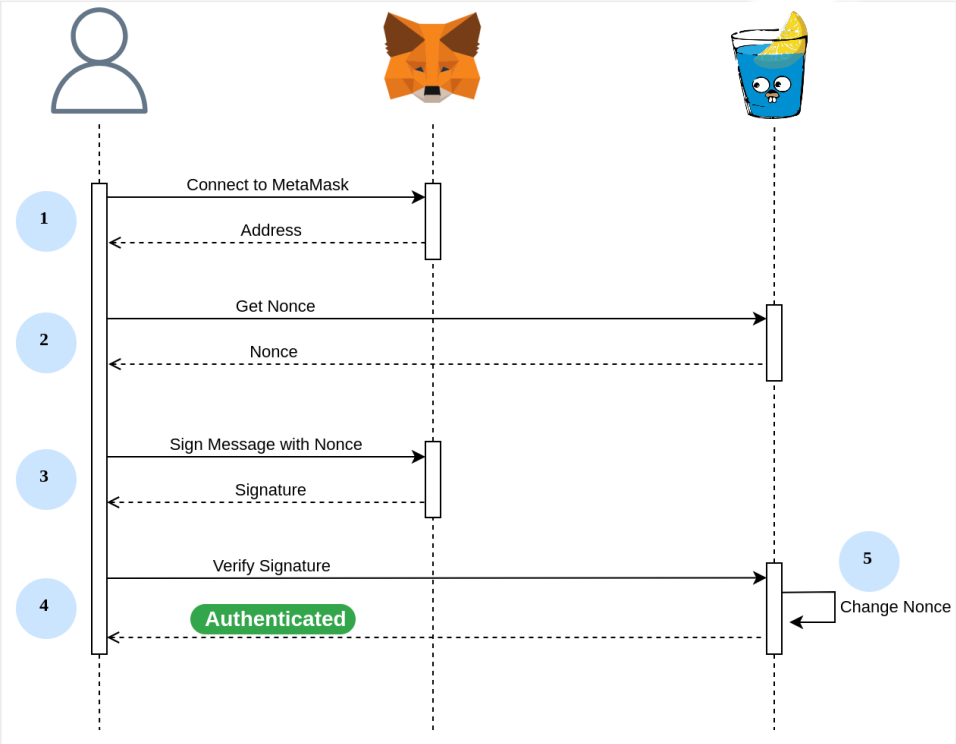


Figura 13: MetaMask login flow

- **/user/home**: questa root richiede un token JWT che, verificato da un middleware, permette l'accesso e indirizza l'utente alla home page del servizio in cui è presente il front end per l'interazione con il contratto. Per contattare questa root è necessario che il token fornito dall'utente contenga al suo interno l'informazione relativa al tipo user base o superiore.

- **/user/app/*ProxyPath**: questa root rappresenta l'aspetto legato al proxy presente all'interno dell'Api Gateway e permette ai client di interagire con il server reale in modo protetto. ***ProxyPath** è una variabile che rappresenta le root reali esposte dal servizio di interazione con la blockchain che però non possono essere direttamente invocate

perchè esposte solamente sulla rete Docker e non sulla rete esterna. Questa root è invocata tramite il front end della home page dell'utente ed è protetta da un Circuit Breaker in quanto risulta essere un punto critico a causa dell'interazione con la blockchain del Servizio reale. Inserendo il Circuit Breaker in questo punto è possibile evitare di far collassare l'intero sistema a causa di un disservizio nelle componenti più interne del sistema.

```
userRoutes := r.Group("/user")

    userRoutes.Static("/css", "../../templates/css")
    userRoutes.Static("/js", "../../templates/js")

    userRoutes.Use(middleware.Authenticate())
    userRoutes.Use(middleware.RoleAuth("user"))
    {
        userRoutes.GET("/user_home", func(c *gin.Context) {
            c.HTML(http.StatusOK, "user_home.html", gin.H{})
        })

        userRoutes.GET("/app/*proxyPath", middleware.Authenticate(),
handlers.ProxyHandler)

    }
```

Figura 14: Architettura

- **/admin/home**: come la precedente effettua i controlli relativi al token e all'utente, in questo caso però il token autorizzerà l'accesso solo agli utenti Admin.
- **/admin/app/*ProxyPath**: come la precedente ma può essere invocata solo dagli utenti autorizzati come Admin.

```
userRoutes := r.Group("/admin")

    userRoutes.Static("/css", "../../templates/css")
    userRoutes.Static("/js", "../../templates/js")

    userRoutes.Use(middleware.Authenticate())
    userRoutes.Use(middleware.RoleAuth("admin"))
    {
        userRoutes.GET("/admin_home", func(c *gin.Context) {
            c.HTML(http.StatusOK, "admin_home.html", gin.H{})
        })

        userRoutes.GET("/app/*proxyPath", middleware.Authenticate(),
handlers.ProxyHandler)

    }
```

Figura 15: Architettura

### 4.1.2 RealServer

Rappresenta il servizio reale invocabile solo tramite il Reverse Proxy. Il RealServer conosce l'indirizzo del contratto e il suo ABI è dunque in grado di interagire con esso (conosce in realtà le informzioni relative al Proxy Contract discusso nell'apposita sezione successivamente). Il RealServer espone due root che prendono input dagli utenti (tramite Proxy) e manipolano il contratto. Esse sono:

- **/registerProduct**: invocata dalla root "/admin/app/registerProduct" del Proxy, prende in input le informazioni relative al chiamante e una descrizione del prodotto che si intende registrare sulla blockchain ed effettua una transazione richiamando il contratto, alla fine ritornerà le informazioni relative alla registrazione come l'id generato, il nome del prodotto, l'indirizzo del creatore e altre informazioni relative alla blockchain di supporto.

- **/getProduct**: invocata dalla root "/admin/app/getProduct" permette dato l'id di interagire con lo smart contract per ritrovare un prodotto corrispondente al codice fornito, se esistente, questa funzione può essere invocata sia dagli utenti base che dagli admin.

### 4.1.3 Smart Contract

Per il deploy dello smart contract su Ganache/Testnet e il testing è stato utilizzato il framework Truffle. Di seguito verranno descritti gli smart contract e le modalità di deploy e testing. Come riportato precedentemente si è scelto di utilizzare il design pattern Proxy per facilitare eventuali cambiamenti dello smart contract in produzione, sono dunque stati sviluppati due differenti contratti:

**ProductLogic**:

Il contratto `ProductLogic` gestisce la registrazione e il recupero delle informazioni sui prodotti. Ecco una descrizione delle sue funzionalità:

**Variabili di Stato e Structs:**

- `owner`: Indirizzo del proprietario del contratto.
- `stopped`: Un'interruzione di emergenza per sospendere alcune funzionalità del contratto.
- `Product`: Una struttura che rappresenta le informazioni di un prodotto, inclusi l'ID, il nome, il produttore e lo stato della registrazione.

**Mapping e Contatori:**

- `products`: Mappa gli ID dei prodotti alle informazioni dei prodotti.
- `productCount`: Conta il numero totale di prodotti registrati nel contratto.

**Eventi:**

- `ProductRegistered`: Viene emesso quando un prodotto viene registrato con successo.
- `EmergencyStopSet`: Viene emesso quando l'interruzione di emergenza viene attivata o disattivata.

**Modificatori:**

- `onlyOwner`: Permette l'accesso solo al proprietario del contratto per alcune funzioni specifiche.
- `whenNotStopped`: Controlla che il contratto non sia in stato di stop prima di eseguire determinate azioni.

**Funzioni Principali:**

- `constructor`: Imposta il creatore del contratto come proprietario e inizializza lo stato di stop come falso.
- `toggleContractStopped`: Consente al proprietario del contratto di attivare o disattivare lo stato di stop del contratto.
- `registerProduct`: Registra un nuovo prodotto nel contratto.
- `getProduct`: Recupera le informazioni di un prodotto registrato.
- `getRegisterProductHash`, `getGetProductHash`, `getTestHash`: Forniscono l'hash delle funzioni specificate, utili per la verifica dei metodi chiamati da altri contratti o servizi.

Questo contratto fornisce un meccanismo per registrare nuovi prodotti, recuperare informazioni sui prodotti registrati e permettere al proprietario di sospendere temporaneamente le operazioni del contratto in caso di emergenza. Di seguito il diagramma UML:

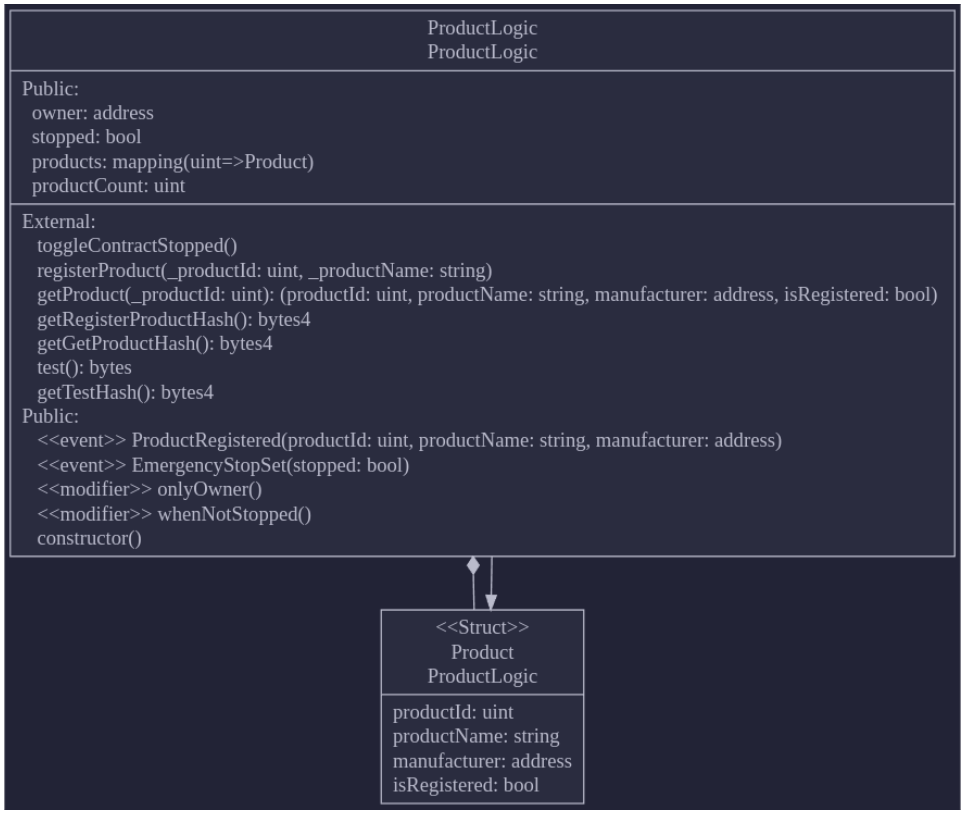


Figura 16: Product Logic UML

### ProductProxy

Il contratto `ProductProxy` è un proxy che funge da intermediario per un contratto logico, consentendo al chiamante di interagire con il contratto logico attraverso questo proxy. Questo contratto ha le seguenti funzionalità:

- `logicContract`: Variabile di stato che memorizza l'indirizzo del contratto logico.
- `constructor`: Alla creazione, imposta l'indirizzo del contratto logico.
- `fallback()`: Funzione fallback che inoltra dinamicamente le chiamate al contratto logico utilizzando la `delegatecall`.

- `receive()`: Funzione di ricezione opzionale per gestire l'invio diretto di Ether al contratto.
- `changeLogicContract()`: Consente di modificare dinamicamente l'indirizzo del contratto logico.

Questo design permette di interagire con il contratto logico attraverso il proxy senza dover conoscere la struttura del contratto logico in anticipo, consentendo anche la possibilità di aggiornare dinamicamente il contratto logico associato al proxy. Di seguito il diagramma UML:

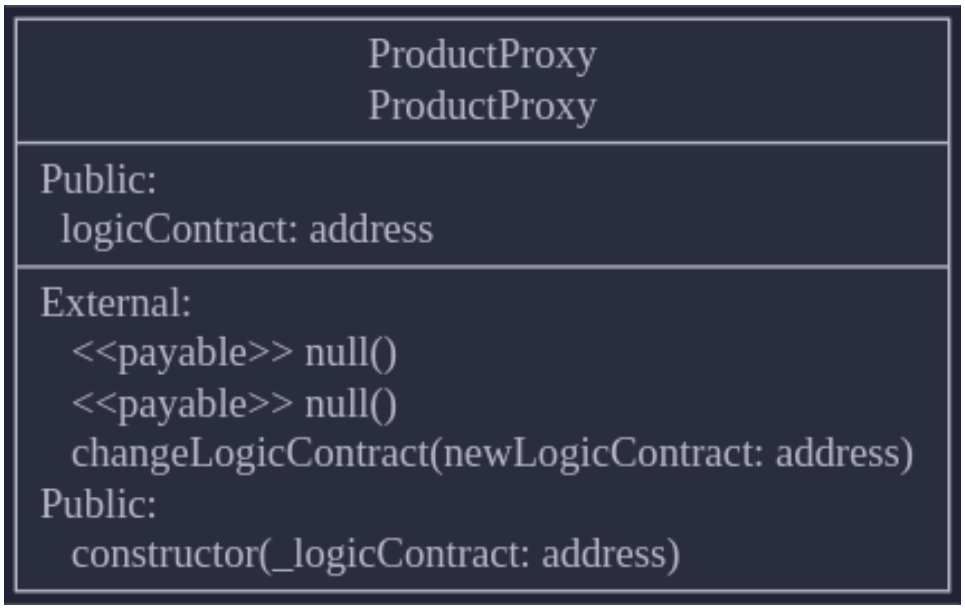


Figura 17: Product Proxy UML

**Truffle Deploy**

Per il deploy dei contratti è stato usato Truffle. Truffle, semplifica il processo di deployment offrendo una struttura ben definita e configurabile tramite il file `truffle-config.js`. Questo file funge da punto centrale per le impostazioni di rete, le configurazioni del compilatore e altre impostazioni essenziali del progetto.

### Configurazione in `truffle-config.js`

Il file `truffle-config.js` orchestr vari aspetti del processo di deployment:

- **Reti**: Agevola la configurazione per diverse reti Ethereum (es. sviluppo, Sepolia, reti private). Qui vengono definite le specifiche di rete come host, porta e ID di rete per effettuare il deployment degli smart contract in diversi ambienti.
- **Compilatori**: Questa sezione specifica la versione del compilatore Solidity utilizzata per compilare gli smart contract. Consente una coerenza nella compilazione attraverso diverse fasi di sviluppo.
- **Mocha**: Parametri rilevanti per i test Mocha, come i timeout, possono essere impostati all'interno di questa sezione.
- **Truffle DB**: Se abilitato, questa sezione configura il database Truffle e le relative impostazioni, tuttavia è consigliabile procedere con cautela poiché la migrazione degli smart contract su Truffle DB non è reversibile.

### Processo di Deployment degli Smart Contract

Il deployment di uno smart contract tramite Truffle coinvolge l'esecuzione di specifici comandi Truffle. Ad esempio, per effettuare il deployment di un contratto chiamato `ProductLogic.sol` su una rete di test Ganache, insieme al relativo script di deployment, viene utilizzato il seguente comando:

```
truffle migrate --network development
```

Questo comando avvia il deployment del contratto `ProductLogic.sol` specificamente sulla rete di sviluppo, come definito nel file `truffle-config.js`. Il

comando `truffle migrate`, associato alla specifica `-network <nome-rete>`, consente di deployare i contratti su una rete mirata. Inoltre, i flag `-f <from>` e `-to <to>` possono essere utilizzati per specificare un intervallo di migrazione, consentendo la migrazione da un file di migrazione specifico a un altro. Se si vuole invece effettuare il deploy su una testnet come Sepolia o la mainnet, sarà necessario impostare il file di configurazione come segue:


Figura 18: Reti truffle per il deploy

Dove $PRIVATE\_KEY$ e la chiave privata dell'account con cui si intende effettuare la transazione verso la blockchain: questo account deve avere un numero di SepEth sufficientemente alto da poter pagare il costo in gas della transazione, per tale motivo è prima necessario collegare l'account ad un **faucet** apposito in modo da ottenere tale quantità di valuta. Invece $PROJECT\_ID$ è l'API KEY ottenuta attraverso **Infura** per lo specifico progetto. Per effettuare il deploy sulla rete Sepolia è possibile lanciare il comando:

```
truffle migrate —network sepolia
```

Si noti che a seconda della rete utilizzata i contratti avranno degli indirizzi differenti che andranno impostati correttamente all'interno del progetto. Se l'operazione è andata a buon fine, truffle darà il seguente risultato:

```
Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.


Starting migrations...
======================
> Network name:    'sepolia'
> Network id:      11155111
> Block gas limit: 30000000 (0x1c9c380)


2_deploy_ProductLogic.js
========================

   Deploying 'ProductLogic'
   ------------------------
   > transaction hash:    0xadd07344d405910ed081754aee88c5a51183876e12ce9e984b0db3b0784fa674
   > Blocks: 0            Seconds: 5
   > contract address:    0x6e525aa41918B0eAc5D7279512e9e43428Cd414A
   > block number:        5008768
   > block timestamp:     1704225420
   > account:             0x1B1AEdd52c5ac647948603f65fbaFaa6B37Fd857
   > balance:             0.2163320575
   > gas used:            1035676 (0xfcd9c)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.02071352 ETH

   Pausing for 2 confirmations...

   ------------------------------
   > confirmation number: 1 (block: 5008769)
   > confirmation number: 2 (block: 5008770)
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.02071352 ETH
```

Figura 19: Deploy con truffle + Infura su Sepolia

**Truffle Deploy Truffle testing**

Il testing con Truffle è un'importante pratica durante lo sviluppo di smart contract su Ethereum. Truffle offre un robusto framework di testing che consente di verificare la correttezza dei contratti intelligenti e garantire che funzionino come previsto.

Truffle utilizza due strumenti principali per il testing:

1. **Mocha**: È un framework di testing per JavaScript che permette di organizzare e eseguire test unitari e di integrazione.

2. **Chai**: È una libreria di asserzioni spesso utilizzata con Mocha per verificare i risultati attesi nei test.

I test vengono scritti utilizzando Mocha e Chai all'interno di file separati all'interno della cartella `test` del progetto Truffle. Questi test possono essere utilizzati per verificare le diverse funzionalità degli smart contract, testare le transazioni e confermare il comportamento atteso. Per eseguire i test, si utilizza il comando `truffle test` dalla directory principale del progetto Truffle. Questo comando avvia l'esecuzione di tutti i file di test presenti nella cartella `test` e visualizza i risultati nel terminale. Nello specifico sono stati scritti dei test per verificare il corretto funzionamento dei contratti e dei design pattern applicati ad essi, l'output ottenuto è il seguente:

```
Compiling your contracts...
===========================
> Compiling ./contracts/ProductLogic.sol
> Compiling ./contracts/ProductProxy.sol
> Artifacts written to /tmp/test--64672-FrFgeZqAkuhI
> Compiled successfully using:
   - solc: 0.8.19+commit.7dd6d404.Emscripten.clang


  Contract: ProductLogic
    ✔ should register a product (192ms)
    ✔ should prevent registering an already registered product (645ms)
    ✔ should retrieve a registered product
    ✔ should toggle emergency stop (124ms)

  Contract: ProductProxy
    ✔ should deploy with the correct logic contract address
    ✔ should change the logic contract address (185ms)
    ✔ should forward function calls to the logic contract via the proxy
    ✔ should revert if logic contract call fails (168ms)


  8 passing (2s)
```

Figura 20: Truffle Test

dove è possibile dedurre dai nomi dei test sviluppati le funzionalità verificate.

### 4.1.4 Data Logger

Il data logger rappresenta un microservizio scritto in Go che sfrutta in modo efficiente il protocollo WebSockets per assicurare una trasmissione in tempo reale dei dati provenienti da NATS. Questa tecnologia avanzata di comunicazione bidirezionale elimina la necessità di continuo polling da parte del client su una specifica dashboard web, consentendo un flusso continuo e istantaneo delle informazioni. Inoltre, il data logger gestisce la scrittura di tali dati su un file di testo. L'utilizzo dei WebSockets non solo ottimizza l'esperienza utente, ma garantisce anche una connettività dinamica e persistente, rendendo la dashboard web reattiva e sempre aggiornata, indipendentemente dalla frequenza di arrivo dei dati da NATS. Tale approccio all'avanguardia garantisce un sistema di monitoraggio robusto e in tempo reale, riducendo significativamente l'impatto dei potenziali malfunzionamenti e garantendo un flusso costante di informazioni affidabili e aggiornate. Come riportato nelle sezioni precedente, vengono riportate sul file di log tutte le risposte alle richieste effettuate dagli utenti e in accordo all'architettura del sistema, gli eventi scatenati dal rate limiter e i cambiamenti di stato del circuit breaker. Di seguito viene riportato un esempio di file di log:


Figura 21: log file

Inoltre collegando una dashboard al container NATS è possibile notare due connessioni attive: l'API gateway e il logger:

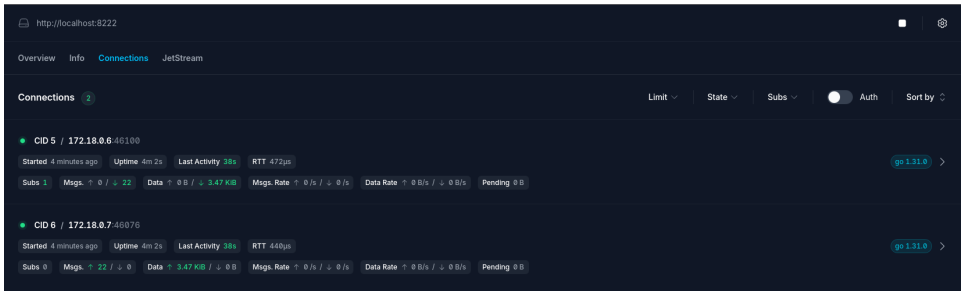


Figura 22: connessioni NATS attive

# 5 User Workflow

In questa sezione, verranno illustrati i passaggi delle operazioni disponibili all'utente attraverso l'interfaccia principale, insieme alle fasi del processo eseguite dal sistema in modo dettagliato

## 5.1 Registrazione

Nel primo passaggio, l'utente si registra utilizzando un modulo dedicato. Durante questo processo, il sistema verifica la correttezza della password inserita e controlla che il nome utente o l'indirizzo MetaMask non siano già stati utilizzati.

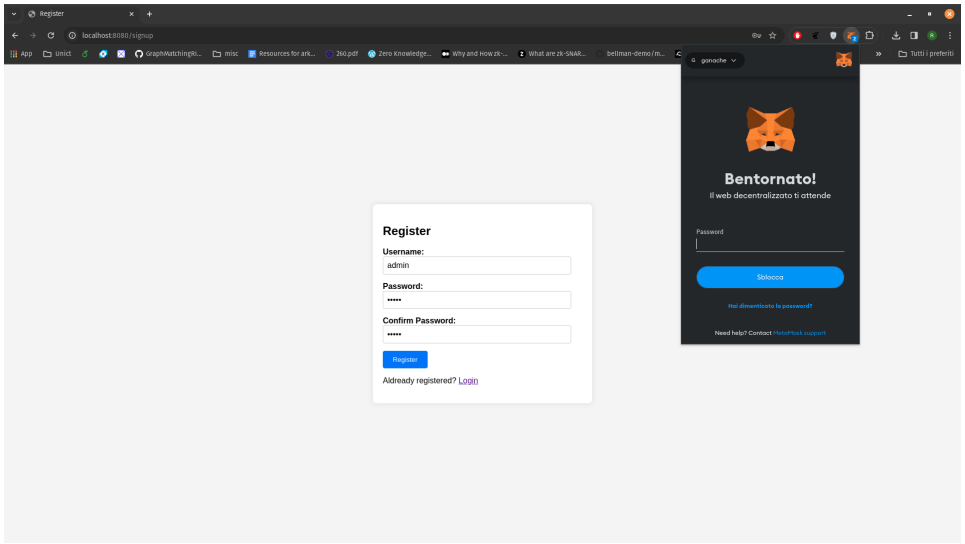


Figura 23: Registrazione utente

Se la registrazione non dovesse andare a buon fine l'utente verrà avvisato attraverso un messaggio di errore:

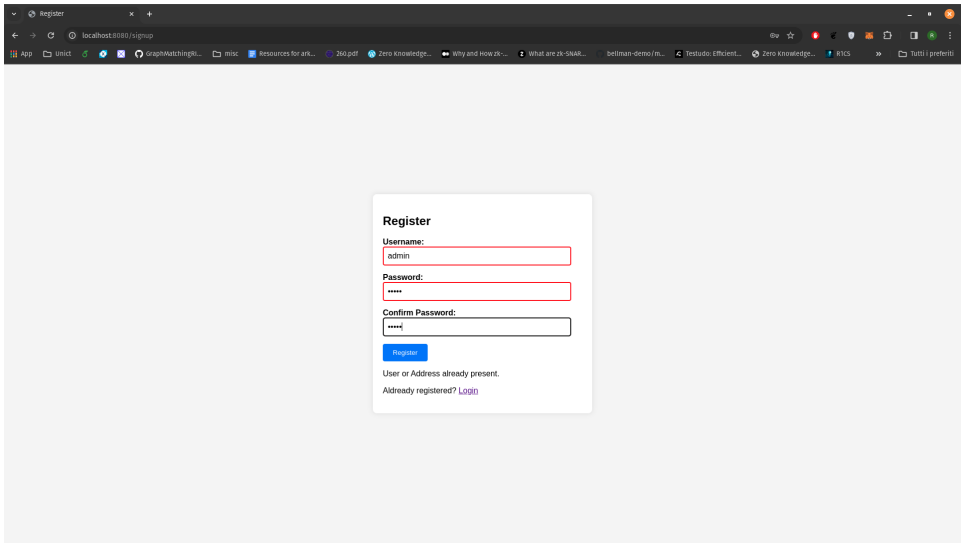


Figura 24: Errore registrazione utente

E' possibile notare che dopo la registrazione, il nuovo utente viene inserito all'interno del database, l'account viene inoltre correlato di una nonce

random che verrà ricalcolata ad ogni login, inoltre la password non è memorizzata in chiaro ma ne viene calcolato un hash tramite apposite librerie.


Figura 25: Database view

## 5.2 Login

Dopo la registrazione, l'utente viene indirizzato alla pagina di accesso, dove può utilizzare le stesse credenziali inserite durante la registrazione. Qui, attraverso il plugin MetaMask, firma una nonce casuale inviata dal server. Se entrambi i processi di accesso, sia il classico accesso tramite il recupero dal database che l'accesso tramite MetaMask, sono completati con successo, il client riceve in modo automatico un token JWT. Questo token viene utilizzato per reindirizzare l'utente alla pagina principale in base al suo ruolo nel sistema. È da notare che la selezione del ruolo avviene in modo trasparente, incorporato direttamente nel token, basandosi su parametri dell'indirizzo usato per l'accesso. Questo approccio unifica l'interfaccia di accesso, nascondendo l'esistenza di una suddivisione dei ruoli nel sistema.

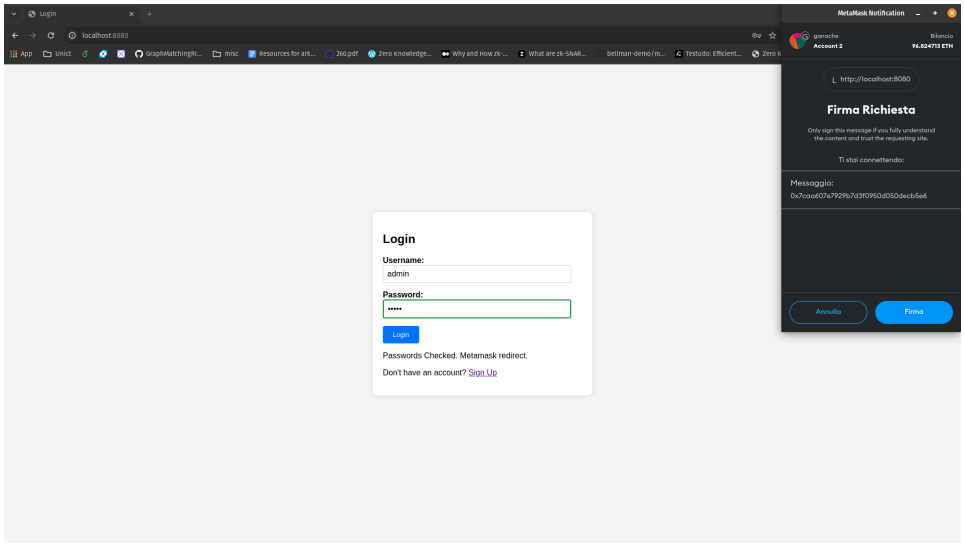


Figura 26: Login utente

Se il login non dovesse andare a buon fine l'utente verrebbe informato attraverso un apposito messaggio di errore:

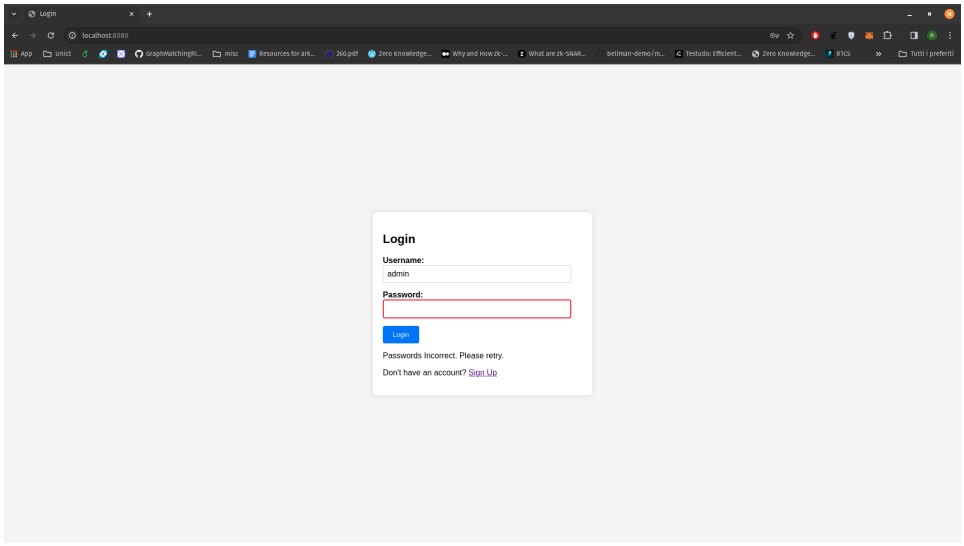


Figura 27: Errore login utente

## 5.3 Admin Home page

Dopo un accesso riuscito, l'utente, in questo caso un amministratore, accede a un'interfaccia dedicata per la gestione degli oggetti. Due pulsanti presenti consentono di accedere al servizio reale, facilitando le chiamate alle principali funzioni del backend che interagiscono direttamente con lo smart contract. È importante sottolineare che tutte queste chiamate transitano attraverso l'API Gateway. In questa schermata, l'utente ha la possibilità di visualizzare le informazioni relative agli oggetti, sia quelli recuperati che quelli appena caricati.

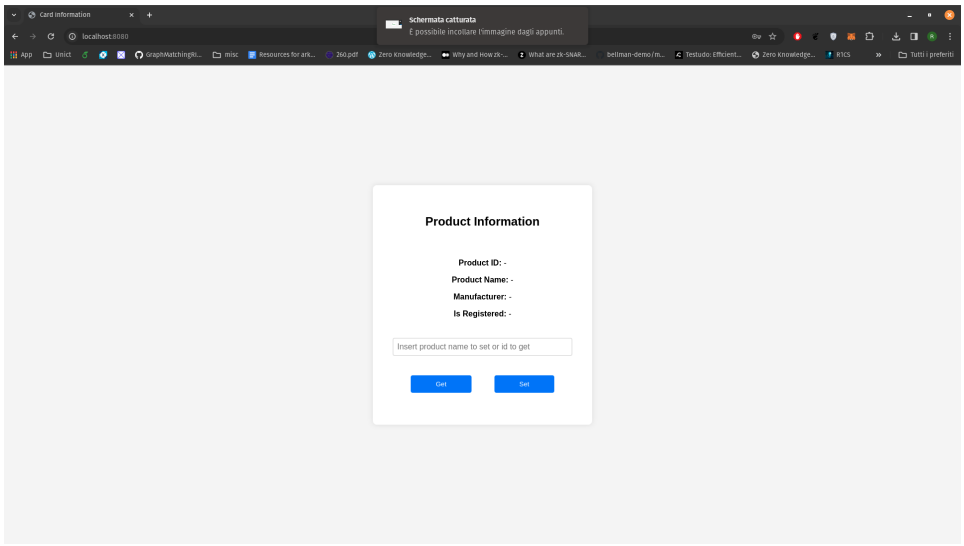


Figura 28: Admin home page

Nel caso di una registrazione del prodotto, l'utente dovrà solamente inserire la descrizione del prodotto che intende registrare, se la procedura va a buon fine verranno mostrate negli appositi campi le informazioni relative alla nuova registrazione. L'id è ottenuto tramite il troncamento di un hash dato dalla concatenazione dell'indirizzo dell'utente e della descrizione del prodotto.

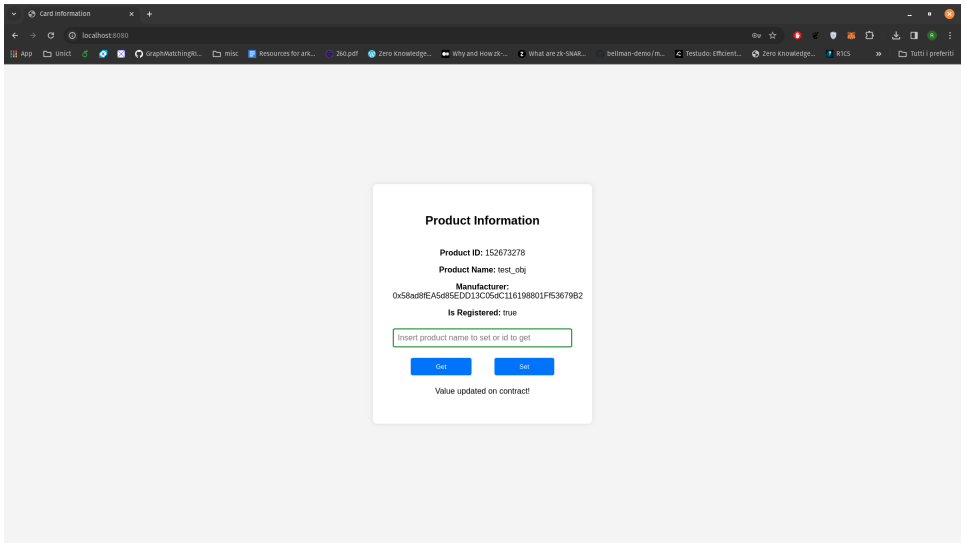


Figura 29: Registrazione prodotto

## 5.4 User Home page

L'home page per gli utenti standard risulta essere simile ma per loro non è possibile effettuare la registrazione di prodotti:

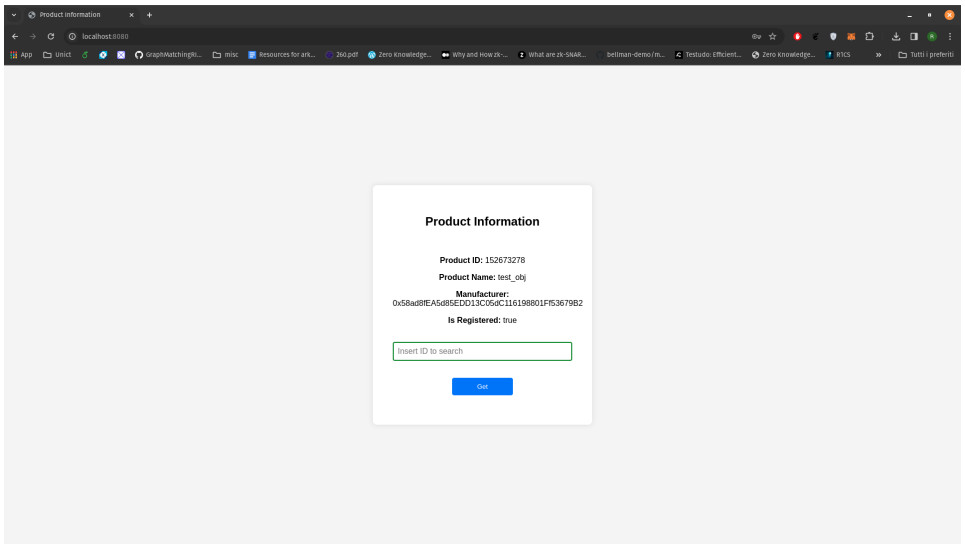


Figura 30: User home page

## 5.5 Stato della blockchain

Ogni volta che si interagisce con un contratto intelligente, sia per leggere (get) che per scrivere (set) dati, si genera una transazione che viene inclusa in un blocco. Queste transazioni vengono eseguite dalle funzioni del contratto e vengono propagate alla rete simulata in Ganache. L'indirizzo dell'utente loggato o dell'account che sta interagendo con il contratto sarà responsabile dell'innesco di queste transazioni. Queste transazioni includono informazioni come l'indirizzo del mittente (utente loggato) e l'indirizzo del contratto a cui è indirizzata la transazione. Quando si effettua il deploy di un contratto, viene generato un indirizzo univoco per quel contratto sulla blockchain. Le transazioni vengono quindi registrate tra gli indirizzi degli account coinvolti (utente e contratto), consentendo la tracciabilità e l'audit delle operazioni effettuate sulla blockchain di test Ganache.

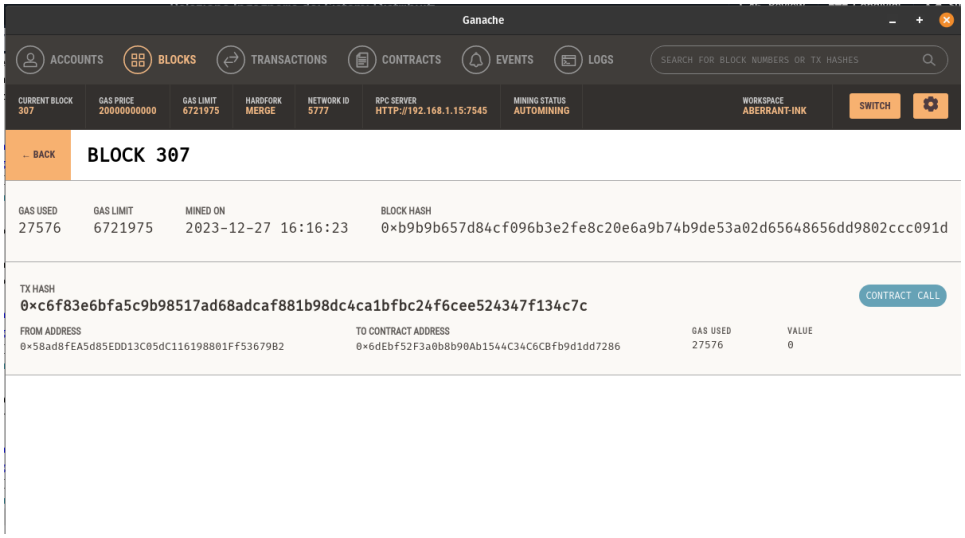


Figura 31: Ganache Tx

## 5.6 Logger

E' possibile collegarsi alla dashboard web per osservare i log in tempo reale:

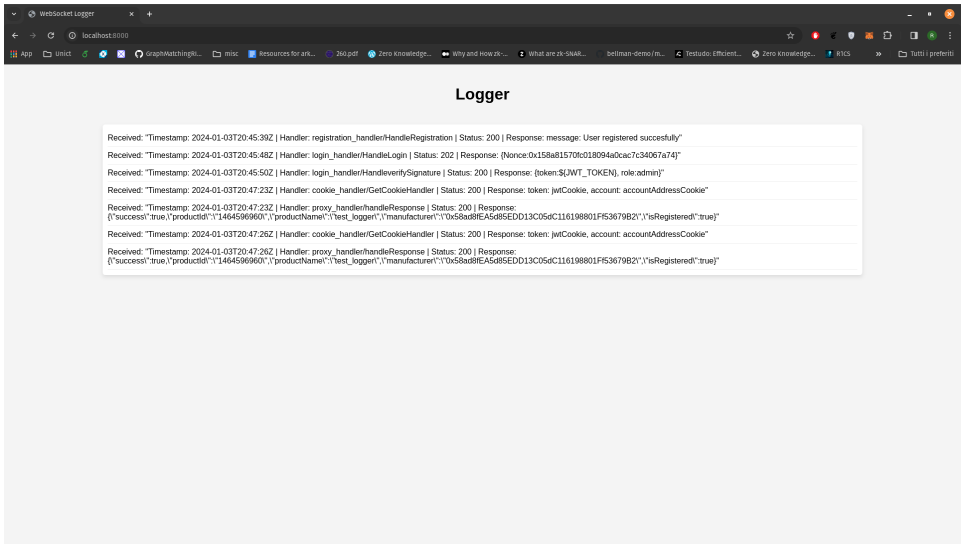


Figura 32: web logging Dashboard

Inoltre, osservando una dashboard collegata a NATS è possibile osservare in tempo reale i log che arrivano sulla coda.

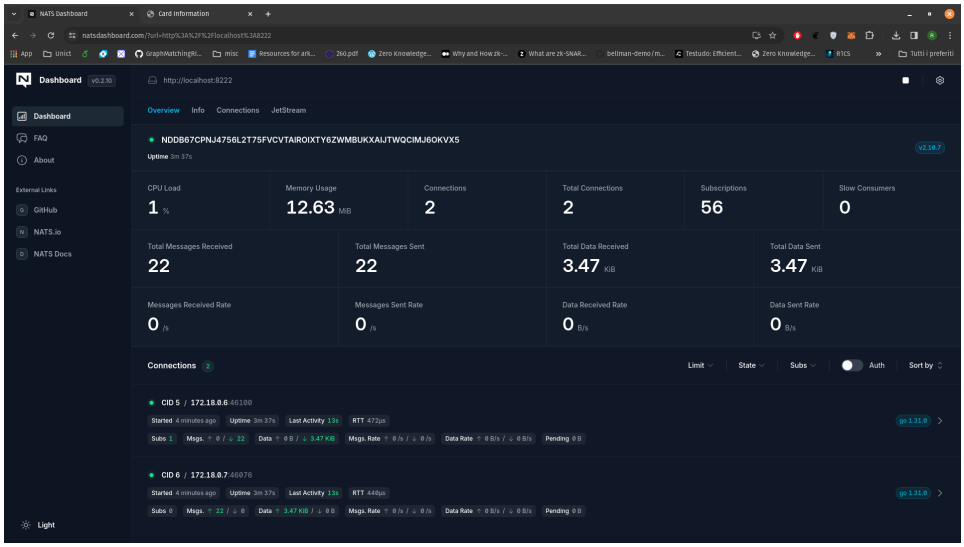


Figura 33: Nats Dashboard

Nella pagina successiva, è presente il diagramma di sequenza che descrive il flusso delle operazioni eseguite, compresi i messaggi di risposta generati dagli strumenti interni al sistema, come il rate limiter e il circuit breaker. Attraverso l'analisi di questo diagramma e dell'interfaccia utente, è possibile seguire con precisione tutte le operazioni eseguite e le interazioni tra il front end e il back end del sistema. Tale rappresentazione consente di comprendere in dettaglio il percorso delle richieste, le risposte ottenute e le varie interazioni tra le componenti del sistema, inclusi i feedback forniti dagli strumenti implementati, come il rate limiter e il circuit breaker.

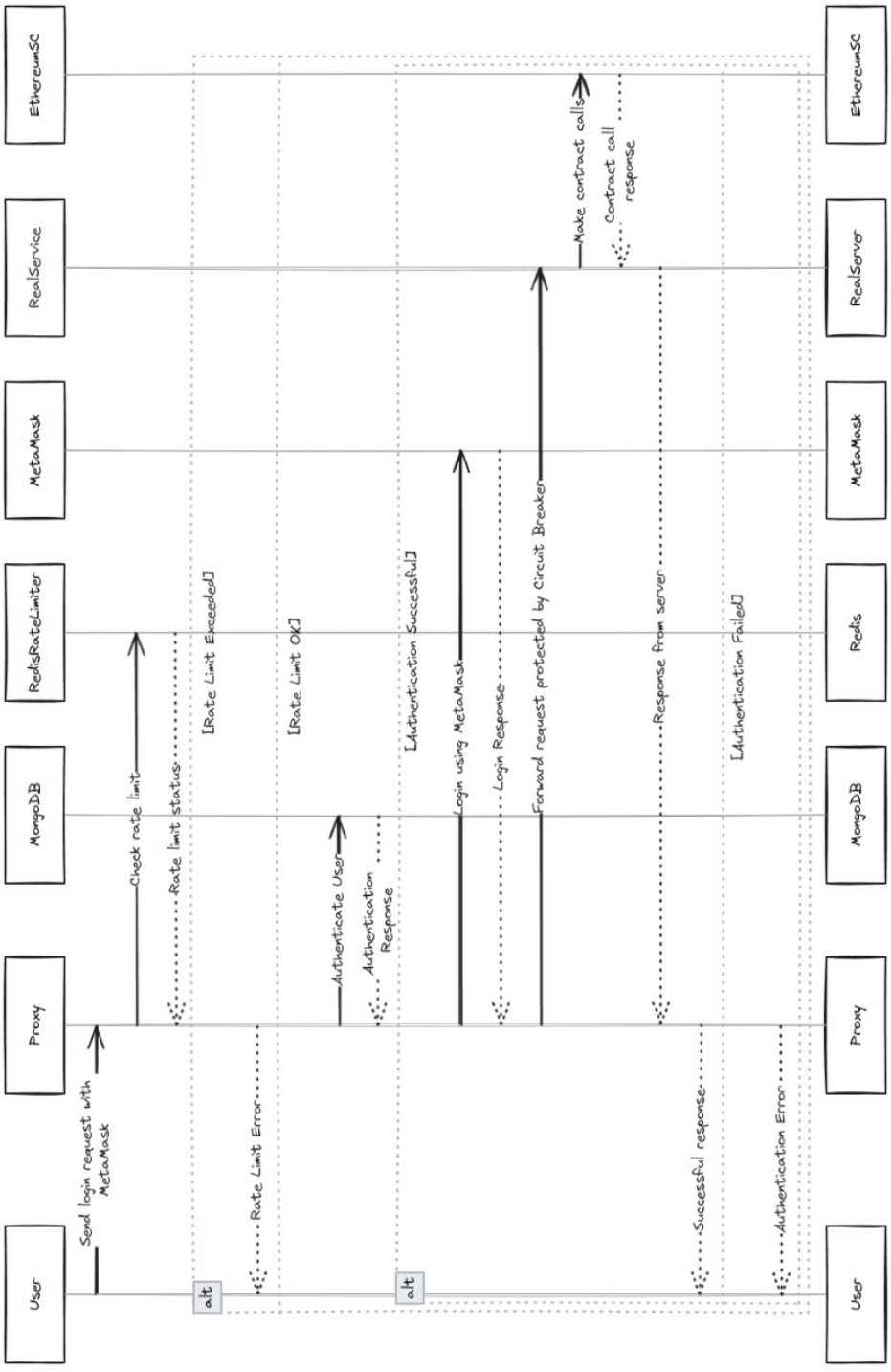


Figura 34: Diagramma di sequenza

# 6 Conclusioni

Il sistema sviluppato rappresenta un esempio di implementazione di una soluzione distribuita e sicura per la gestione e la convalida dei prodotti sfruttando la tecnologia blockchain. Tuttavia, pur essendo un'architettura funzionale, ci sono opportunità per ulteriori miglioramenti. L'utilizzo dell'architettura dei microservizi e l'orchestrazione tramite container Docker forniscono flessibilità e gestione semplificata dell'ambiente di sviluppo. L'API Gateway, il nucleo di questa soluzione, gestisce l'autenticazione multipla degli utenti e implementa un efficace controllo degli accessi basato sui ruoli. La persistenza delle informazioni degli utenti su MongoDB e l'utilizzo di Redis come servizio di caching e "rate limiter" migliorano le prestazioni complessive del sistema. L'interazione con la blockchain attraverso Web3.js e lo smart contract Solidity su una testnet sviluppata tramite Ganache o Sepolia tramite l'uso di un nodo Infura, è un punto di forza per la registrazione e la validità dei prodotti. Tuttavia, esistono opportunità per ottimizzare ulteriormente questa comunicazione e rendere più efficienti le operazioni con la blockchain. Il processo di sviluppo e testing utilizzando Truffle ha fornito un ambiente affidabile, ma potrebbero essere esplorate altre metodologie o strumenti per migliorare ulteriormente la qualità e la robustezza del codice. L'uso di NATS correlato a websocket per l'instradamento delle chiamate all'API Gateway e l'implementazione di un microservizio dedicato alla registrazione delle attività sono buone pratiche che garantiscono una tracciabilità accurata delle azioni nel sistema. Tuttavia, potrebbe essere valutata l'introduzione di soluzioni di monitoraggio e gestione delle performance per ottimizzare ulteriormente il sistema nel suo insieme Come Jaeger e OpenTracing. Un vantaggio significativo di questa implementazione risiede nella presenza dei pattern specifici già

integrati negli smart contract. Questi pattern si rivelano cruciali in situazioni di patching, consentendo modifiche e aggiornamenti senza richiedere una ridefinizione completa del progetto. Tale caratteristica garantisce la capacità di apportare correzioni o miglioramenti in modo trasparente agli utenti di un sistema in produzione, evitando interruzioni sostanziali durante l'evoluzione del sistema stesso. In conclusione, "ValidiVault" rappresenta un sistema distribuito e sicuro per la gestione dei prodotti tramite blockchain, ma esistono spazi per ulteriori miglioramenti che possono essere agevolati dall'architettura già progettata. Investire in ottimizzazioni della comunicazione con la blockchain, esplorare nuove metodologie di sviluppo e considerare soluzioni avanzate di monitoraggio potrebbe portare a un sistema ancora più efficiente e performante.